Sigma Intelligence

Lecture 2001

To our lovely Freshmen…

♦ Index ♦

♦ Introduction ♦

sigma 신입생 강의에 오신 여러분 환영합니다. 이번 강의는 새로 Sigma Intelligence의 회원이 되신 여러분들을 대상으로 한 달 동안 진행됩니다. 강의 시간은 모두 아시죠? 매주 목요일 저녁 6시부터 8시까지 진행될 예정입니다.

이번 강의에서 다룰 주제는 크게 다음과 같습니다. 우선 여러분이 가장 많이 접하게 되는 PC의 기본적인 작동 원리와 OS의 개략적인 기능에 대해 살펴보겠습니다. 그리고 80KC196을 중심으로 micro controller에 대해 알아보게 됩니다.

여러분께 알찬 강의가 되도록 노력하겠습니다.

1. Fundamentals of Computer

1.1. Computer

백과 사전에 이렇게 쓰여 있더군요..

프로그래밍이 가능한 전기적 장치로서 입력 자료를 받아들여서 처리하여, 그 처리된 정보를 저장하고 검색하여 결과를 출력하는 일을 하는 기계.

여러분이 항상 접하는 컴퓨터가 어떠한 원리로 동작하는 지 생각해 보신 적 있으세요? 혹시 컴퓨터를 뜯어보고 우리 전기공학부에 오고 싶다는 생각을 한 분은 없으신지요..

컴퓨터는 크게 CPU(Central Processing Unit)과 메모리, I/O (Input/Output) Device로 나눌 수 있죠. CPU는 BUS라는 여러 가닥의 전선을 통해 메모리와 I/O 장치들과 연결됩니다. 자세한 동작 원리는 제 4 강에서 배우도록 하고요. 이번 시간에는 개략적인 동작 원리와 단순한 hardware에 불과한 computer를 동작시키는 Operating System에 대해서 알아보겠습니다. Windows에서 여러 개의 program이 동시에 돌아갈 수 있는 이유가 무엇인지 궁금하지 않았나요?

1.2. CPU 란?

1.2.1. CPU의 종류

CPU란 **C**entral **P**rocessing **U**nit의 약자 입니다. 우리말로는 중앙 연산 처리 장치라고 합니다.

Micro Processor의 핵심 소자로 사람으로 치면 두뇌에 해당한다고 할 수 있죠. CPU의 내부 구조는 크게 **ALU**(**A**rithmetic **L**ogic **U**nit, 산술 논리 연산 장치)와 **CU**(**C**ontrol **U**nit)로 나눌 수 있습니다. 또한 외부적인 구분으로 Address bus와 data bus, control bus등을 가지고 있죠. 이것들은 CPU의 기본 동작인 fetch(memory에서 data을 읽어오는 일)와 해독, 실행을 위해 필요한 요소라 할 수 있습니다. 대부분의 CPU는 전원이 들어오면 0000H번지의 내용을 fetch하고 프로그램을 수행하게 됩니다. 그래서 대부분의 시스템의 ROM의 첫 기억 장소가 0000H번지로 지정되어 있죠.

1) CISC (complex instruction set computer)

CISC는 필요한 모든 명령어 셋을 갖추도록 설계된 마이크로프로세서를 일컫습니다. 가능한 모든 경우에 해당하는 수백 개의 명령어를 갖는 CPU를 의미하는데요. 그러나 그 후, 명령어 셋 자체를 가장 자주 사용되는 명령어만으로 개수를 줄임으로써 대부분의 활용업무 면에서 소요되는 시간을 줄일 수 있는 방법이 고안되었는데, 이것을 RISC (reduced

instruction set computer)라 합니다.

IBM의 System/6000 기종과 매킨토시 컴퓨터에 사용되고 있는 PowePC라는 마이크로프로세서가 바로 RISC 형태의 프로세서이며, 펜티엄을 포함한 인텔 계열의 모든 프로세서는 CISC 프로세서이다. RISC는 CISC의 길고 복잡한 명령어 설계를 가져 다 짧고, 처리가 빠른 여러 개의 명령어로 바꾸었다.

2) RISC (reduced instruction set computer)

RISC에 대해 자세히 알아볼까요? RISC는 적은 수의 컴퓨터 명령어를 수행하도록 설계된 마이크로프로세서로서, 좀더 빠른 속도로 동작될 수 있습니다. 컴퓨터는 수행해야 할 각 명령어 형태마다 트랜지스터와 회로들이 추가되어야 하기 때문에, 많은 수의 컴퓨터 명령어 셋을 갖춘 마이크로프로세서는 만들기도 복잡하고, 실행속도도 늦은 경향이 있죠. - 한 싸이클 당 한 개의 명령 수행

- load/store 명령만이 메모리 접근. 다른 명령은 레지스터만을 이용
- 주소 지정 방식이 단순함
- 제어 유닛의 단순화
- 고속의 임의논리회로 *random logic*에 의한 제어
- 단순한 명령 셋
- 일정한 명령 길이
- 고정된 필드 위치
- 소수의 명령 양식(포맷)
- 범용 레지스터의 수가 많음

3) Intel micro processor

밑에 Intel micro processor의 발전 과정을 적어보았습니다. 클럭 속도가 빨라지고, 레지스터, 버스의 길이가 늘어났다는 걸 한 눈에 알아볼 수 있죠?

인텔 마이크로프로세서 계보

모 델	8085	8086	8088	80286	80386	80486	펜티엄	펜티엄 프로
생산 연도	1976	1978	1979	1982	1985	1989	1992	1995
클럭 주파수(MHz)	8	10	8	16	33	50	66	150
트랜지스터 수	6500	29000	29000	130000	275000	120 만	310 만	550 만
물리 메모리	64K	1M	1M	16M	4G	4G	4G	64G
내부 데이터 버스	8	16	16	16	32	32	32	32
외부 데이터 버스	8	16	8	16	32	32	64	64
어드레스 버스	16	20	20	24	32	32	32	36
데이터 크기(bit)	8	8,16	8,16	8,16	8,16,32	8,16,32	8,16,32	8,16,32

클럭이 빨라졌다는 것은 한 명령어를 수행하는 시간이 짧아졌다는 것을 의미하고 레지스터의 길이와 버스의 길이가 늘어났다는 건 한 번에 더욱 많은 정보를 처리할 수 있게 되었다는 것을 의미합니다.

좀 더 자세히 살펴보죠.

8086은 16bit의 레지스터 용량을 지닙니다. 즉, 기본적인 저장 공간의 크기가 16bit로 이루어져 있다는 뜻이 됩니다. 이러한 레지스터로 나타낼 수 있는 메모리의 주소는 2^{16} = 64KB 로 제한되겠죠. 하지만 이걸로는 많은 메모리를 사용할 수 없기 때문에 다른 방법이 쓰입니다. 일단 메모리의 주소는 두 개의 변수로 지정됩니다. Base address와 offset address라고 하는데요. base address * 16 + offset address 가 실제 메모리의 주소가 되는

겁니다. 예를 들어 base address = 0x1234 이고, offset address = 0x5678 이라면, 실제 메모리 주소는 0x12345 + 0x5678 = 0x179bd 가 되는 겁니다. 이런 방식을 이용해서 2^{20} = 1MB 의 주소를 지정할 수 있게 됩니다. 20개의 어드레스 버스를 이런 방식으로 지정할 수 있습니다.

CPU 가 발전함에 따라 레지스터의 용량도 계속 늘어났고, 당연히 사용할 수 있는 메모리의 용량도 따라 증가했죠.

1.2.2. CPU의 구성

산술논리 연산장치 arithmetic and logic unit, ALU
제어장치 control unit
프로그램 카운터 program counter, PC
상태 레지스터 status register
메모리 주소 레지스터 memory address register, MAR
메모리 데이터 레지스터 memory data register, MDR
명령 레지스터 instruction register, IP
범용 레지스터 general-purpose registers

컴퓨터에서 데이터 처리 동작을 수행하는 부분을 CPU 라고 하죠. 레지스터는 이진 신호를 일시적으로 기억하며, 필요한 경우에는 read와 clear할 수 있는 회로이죠.

*** Stack & Queue**

Stack은 LIFO (last-in, first-out) 이고, Queue는 FIFO (first-in, first-out)입니다. 무슨 말인고 하니 LIFO는 가장 최근에 요청된 것을 가장 먼저 처리하는 방식으로서, 가장 처음에 들어온 요청은 최후에 처리된다. 비록 LIFO가 일견 공정하지 않은 듯 하지만, 경우에 따라서는 더 효율적일 수도 있죠. 예를 들어 stack을 이용해서 수식 계산을 할 수도 있고, 함수의 return address를 저장할 수도 있습니다.

그에 반해 FIFO는 queue로부터 프로그램의 작업요청을 처리하는 방식으로서, 가장 오래된 요청(가장 먼저 요청된)을 가장 먼저 처리하죠. 쉽게 생각해서 선착순이라고 보면 됩니다.

1.3. I/O Device

1.3.1.저장 매체의 접근 시간

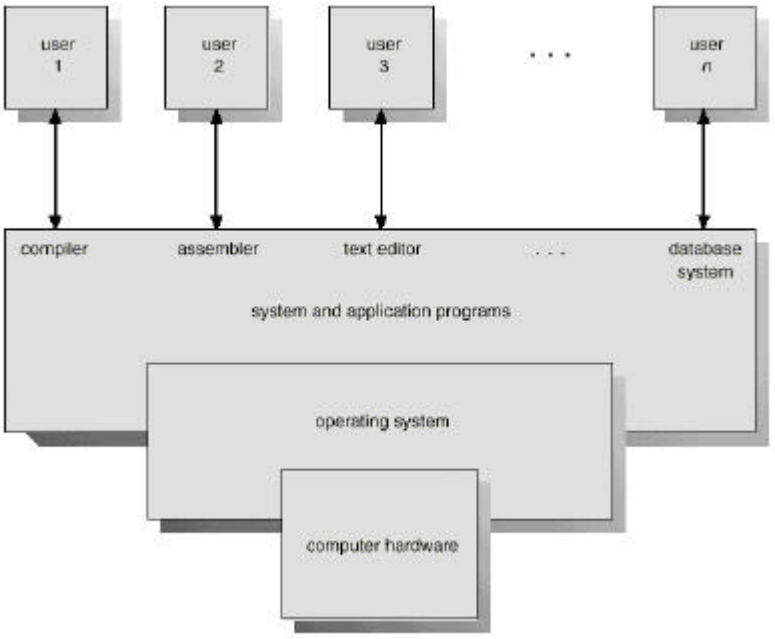
저장매체	일반적인 접근시간	데이터 처리율
CPU 레지스터		
온칩 캐쉬		
캐쉬 메모리	10-30 nsec	
주기억장치	50-100 nsec	
디스크 캐쉬	70-500 nsec	
하드 디스크	10-50 msec	600-6000 KB/sec
플로피 디스크	95 msec	100-200 KB/sec
CD-ROM	100-600 msec	150-1000KB/sec
테이프	0.5 sec 이상	5-20 KB/sec (cartridge)

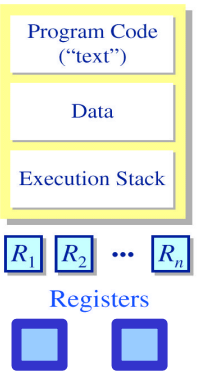
저장 매체의 접근 시간이 큰 차이를 보이는 걸 볼 수 있죠? 가장 흔히 말하는 메모리, 즉 주기억 장치와 하드 디스크의 속도 차이가 얼마인지 한 번 계산해 보세요. 자그마치 10만 배 이상이랍니다. 당연히 하드 디스크를 읽는 것은 컴퓨터에게는 매우 지루한 작업임이 틀림없겠죠? 그래서 하나 이상의 process를 운영할 때에는 시간 낭비를 줄이기 위해 OS에서 process scheduling을 한답니다. 그럼, 자세히 배워볼까요?

1.4. Operating Systems

1.4.1. operating system 이란?

운영체제는 한정된 컴퓨터 시스템의 자원을 보다 효율적으로 관리하고 운영함으로써 사용자들에게 편의를 제공하고자 하는 시스템 프로그램으로 인간과 기계간에 interface 역할을 합니다.운영체제의 주요한 목적은 convenience와 efficiency로 요약할 수 있죠.





1.4.2. 운영체제의 기능

Processes, CPU scheduling, Process synchronization

Memory management, Virtual memory

File system, I/O system, Secondary-storage system

1.4.3. Processes

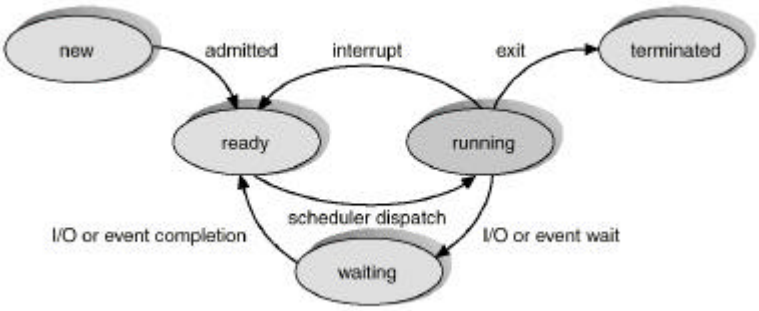
1) Process

Process는 실행 중인 프로그램으로 정의됩니다.

Process는 Program code, Data, Execution stack, CPU Registers, Program counter (PC), Stack pointer (SP) 등을 포함합니다.

2) Process State

Process는 실행 상태에 따라 다음과 같이 분류됩니다.



- new: The process is being created.
- running: Instructions are being executed.
- waiting: The process is waiting for some event (e.g., I/O completion) to occur.

- ready: The process is waiting to be assigned to a processor.
- terminated: The process has finished execution.

1.4.4. CPU Scheduling

1) Basic Concepts in Scheduling

- Non preemptible: 한번 실행된 process는 그 실행이 종료될 때까지 중간에 임의로 멈출 수 없습니다.
- Preemptible: process를 중간에 멈출 수 있습니다.

2) CPU Scheduling

Multiprogramming : CPU의 효율 (utilization, throughput)을 높이기 위해서 실행하고자 하는 여러 개의 process에게 골고루 CPU를 분배하는 것을 말합니다. CPU Scheduling에 의해 어떤 process에 얼마만한 CPU 사용 시간을 분배할 것인지를 결정하게 됩니다.

♦ CPU scheduling algorithm 에서 고려해야 할 것들.
- CPU 중심으로 본다면 utilization과 throughput을 가능한 높여야겠죠?

Process 중심으로 본다면 Turnaround time, Waiting time, Response time을 가능한 줄여서 빠른 응답을 얻어야겠죠?

3) Examples of CPU scheduling Algorithms

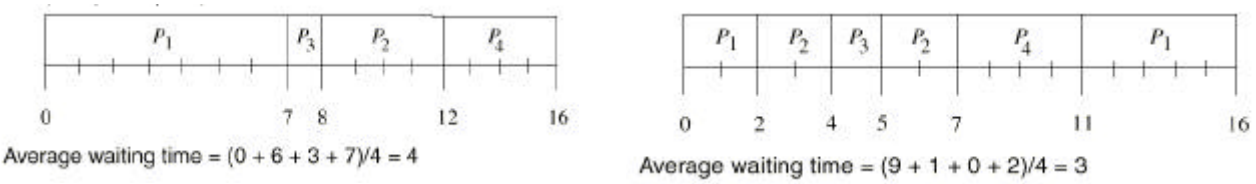
¨ **First-come First-served (FCFS)**

ready queue에 빨리 들어온 순서대로 실행됩니다.
Non preemptive입니다. 즉 앞의 process가 끝난 뒤에 다음 process가 실행됩니다.
문제점들..
만약 오래 걸리는 process 뒤에 짧은 process 가 기다리고 있다면 뒤에 기다리는 시간이 너무 길어지게 됩니다.

♦ Shortest Job (Request) First (SJF)
예상되는 CPU 사용시간을 예측하여 가장 빨리 끝나게 되는 Process 에 우선권을 줍니다.
preemptive 와 non preemptive 두 가지 경우가 있는데요..
Optimizes average waiting time로 가장 이상적이라고 할 수 있죠.

- Problems : 예상 CPU 사용시간을 측정하기 어렵죠.
- Starvation: 오랜 CPU 수행시간을 필요로 하는 process 는 무한정으로 기다릴 수 있습니다. 그것을 starvation 이라 합니다.
- < example >

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4



a) Non-Preemptive SJF

b) Preemptive SJF

♦ Round Robin (RR)

process는 FIFO 의 대기 큐에 적재되지만, 주어진 시간 할당량(CPU time slice, quantum) 안에 작업을 마쳐야 하며, 할당량을 다 소비하고도 작업이 끝나지 않은 process 는 다시 대기 큐의 맨 뒤로 되돌아가게 됩니다.

따라서 Preemptive 이면서 시스템이 사용자에게 적합한 응답시간을 제공해 주는 대화식 시분할 시스템에 적당합니다.

- Problems : 잦은 switching 때문에 overhead가 클 수 있습니다.

1.5. Memory management

컴퓨터가 자료를 처리하려면 그 자료와 이를 처리할 수 있는 프로그램을 저장할 수 있는 곳이 있어야 하죠. 처리장치로 프로그램을 불러내어 처리하는 과정에서 일시적인 저장이 필요하기도 하고, 또한 지금 바로 처리장치가 필요로 하지 않는 자료와 프로그램을 얼마 동안 보존하였다가 필요할 때 꺼내서 사용할 수 있는 저장장치도 필요합니다. 이러한 저장 기능을 담당하는 장치를 통털어 기억장치라 일컫습니다.

기억장치는 읽고 쓰는 속도에 비례하여 가격이 비싸기 때문에, 처리장치와 직접 연결된 기

억장치는 작은 용량의 속도가 빠른 것으로 구성하여 처리 속도를 높이고, 대량의 자료 저장을 위하여는 속도는 느리지만 가격이 싼 것으로 구성하여 비용과 성능을 감안하여 컴퓨터 시스템을 구성하는 것이 대부분입니다. 이처럼 한 시스템에서 여러 종류의 기억장치를 연결하여 속도와 저장량을 보완한 구조체를 메모리 계층이라 합니다. 메모리 계층은 크게 3단계로 구분할 수가 있죠.

첫째, 중앙처리장치에 내장되는 기억장치로서 처리장치와 속도가 비슷한 플립플롭으로 이루어진 레지스터들을 말하며, 이들은 명령이나 자료들을 일시적으로 저장하는 역할을 하며 버퍼(buffer) 또는 캐시(cache) 메모리라고 합니다.
둘째, 주기억장치로서 컴퓨터가 작동하는 동안 처리장치가 필요로 하는 일이 많은 프로그램 명령어와 자료를 저장하고 있는 기억장치입니다. 비교적 용량이 크고 속도도 빠르죠. 이 장치의 특징은 중앙처리장치 명령에 의하여 기억된 장소에 직접 접근하여 쓰고 읽을 수 있습니다. 여기에 사용되는 소자에는 램(RAM)을 비롯하여 여러 가지 소자들이 있는데요. 일반적으로 단순히 기억장치라 하면 바로 이 주기억장치를 의미합니다.

셋째, 보조기억장치로서 용량은 크지만 기계적인 부품들이 연결되기 때문에 속도는 주기억장치에 비하여 상당히 느립니다. 이는 입력되는 자료와 처리결과를 보관하거나, 중앙처리장치에서 계속적으로 필요로 하지 않는 각종 프로그램들을 저장하는데 사용합니다. 가장 대표적인 것이 보통 하드 디스크라 불리우는 자기 디스크입니다.

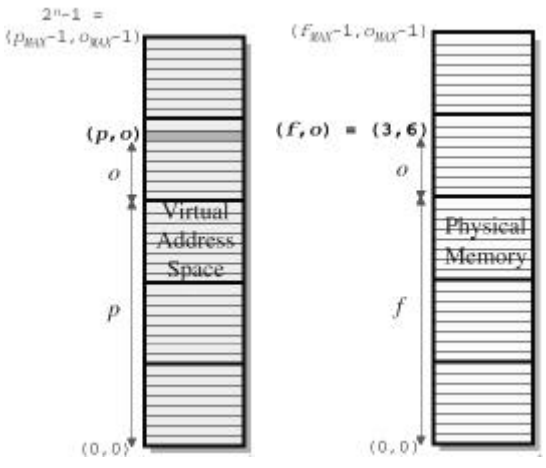
그럼 우선 컴퓨터의 성능에 가장 큰 영향을 미치는 주기억 장치 (memory) 에 대해 알아보죠.

1.5.1. Virtual Memory Concept

memory는 크기가 한정되어 있습니다. 하지만 운영체제 위에서 수행되는 각각의 응용프로그램들은 자신만이 CPU를 독점하고 있으며, 무한히 많은 memory를 가지고 있다고 착각하게 됩니다. 즉 memory의 물리적 측면은 user에게는 숨겨져 있는 것입니다.

Memory is a logically unbounded *virtual (logical) address space* of 2^n bytes.

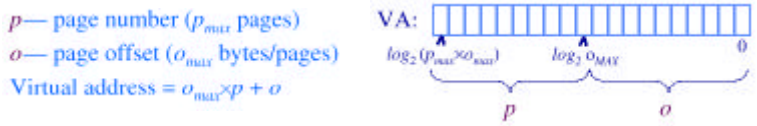
Only portions of virtual address space are in physical memory at any one time.



1.5.2. Paging

process의 virtual address space는 *page* 라 불리는 일정한 크기의 블록들로 나누어져 있습니다.

virtual address 는 (*p*, *o*). 의 쌍으로 표현됩니다.



실제의 memory는 *frame* 이라 불리는 일정한 크기의 블록들로 나누어져 있습니다.

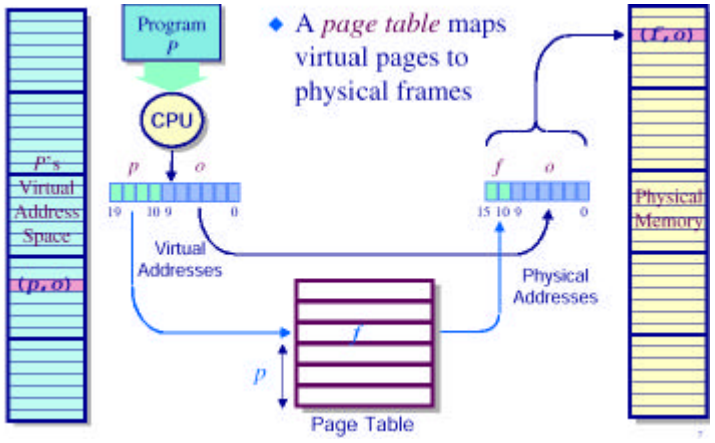
size of page = size of frame

실제 memory address는 *(f, o)*. 의 쌍으로 표현됩니다.

Page는 각각 임의의 어느 frames에 대응됩니다.

virtual address에서 Page들은 연속적이지만, 실제 memory에서 대응되는 frame들은 임의로 배열되어 있고, 일부의 page만이 frame으로 대응됩니다.

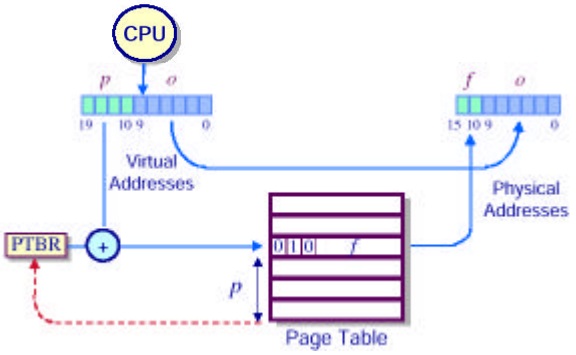
a) Paging: Virtual address translation



b) Paging: Page Table Structure

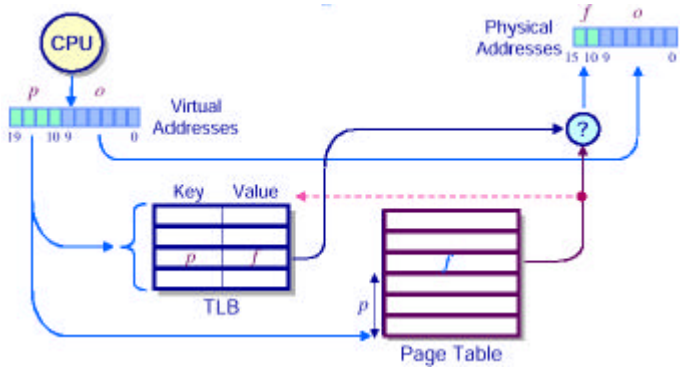
각 process 마다 고유한 Page Table이 존재합니다.

Page Table에는 해당하는 page에 대응되는 frame의 물리적 주소가 적혀 있습니다.



c) Paging: Translation Lookaside Buffer

Page Table을 이용해서 frame에 접근하려면 두 번의 memory access 가 필요합니다. 이건 커다란 낭비가 되죠. 그래서 Translation lookaside buffer 를 이용해서 자주 사용하는 page 들은 한 번의 memory access로 address를 알 수 있도록 하였습니다.



d) Page Faults

Page에 없는 내용을 참조하려고 하면 내용을 찾을 수 없게 되겠죠. 그래서 memory에서 필요 없는 내용을 버리고 disk에서 필요한 내용을 memory로 불러들여야 하겠죠. 이것을 page fault 라고 합니다.

1.5.3. Page replacement Algorithms

a) 가장 이상적인 Algorithm

앞으로 가장 오랜 시간동안 사용되지 않을 page를 교체하면 되겠죠.

그러나 앞으로 무엇을 사용할지를 알아내는 것은 거의 불가능한 일이므로 희망사항일 뿐인 algorithm입니다.

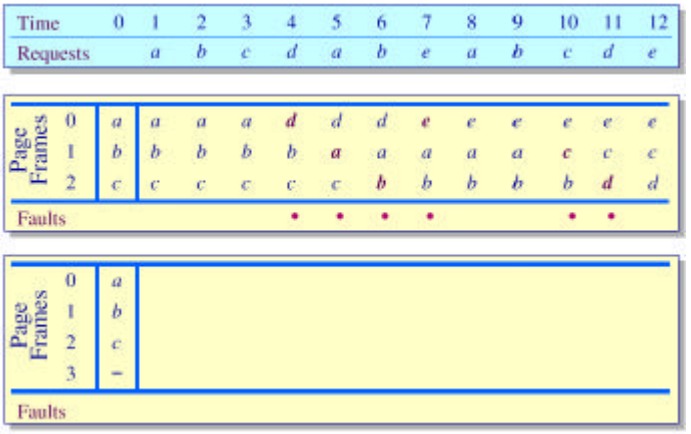
b) FIFO Algorithm

가장 오래된 것을 교체합니다.

가장 단순한 방법이죠. 하나의 queue만 있으면 되니까요.

Belady's Anomaly

More frames ≠ Better performance for the FIFO algorithm



c) LRU (Least Recently Used) Algorithm

가장 오래된 것을 교체하는 겁니다.

Maintain a stack of recently used pages according to the recency of their uses.

Top: Most recently used (MRU) page.

Bottom: Least recently used (LRU) page.

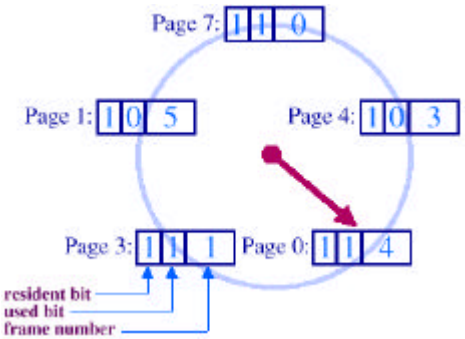
Always replace the bottom (LRU) page.

d) LRU Approximation - Second-Chance Algorithm

clock algorithm 이라고도 불립니다. UNIX 에서 쓰이는 방법으로, Memory에 원형의 page list를 둡니다.

매번 참조를 할 때마다 해당 page의 reference (also called used or clock) bit 을 1 로 set 하고, page fault가 발생하면, clock을 돌리면서 reference bit = 0 인 page 를 찾죠.

clock 이 한 바퀴를 도는 동안에 한 번도 참조된 적이 없는 page는 reference bit = 0 이겠죠? 그 page를 교체하는 겁니다.




```
func Clock_Replacement
begin
  while (victim page not found) do
    if(used bit for current page = 0) then
      replace current page
    else
      reset used bit
    end if
    advance clock pointer
  end while
end Clock_Replacement
```

2. 논리 회로와 CPU

2.1. About Digital

2.1.1. Digital & Analog

논리 회로에 들어가기 앞서 간단히 Digital과 Analog의 의미와 그 차이점에 대해 알아보도록 하겠습니다.

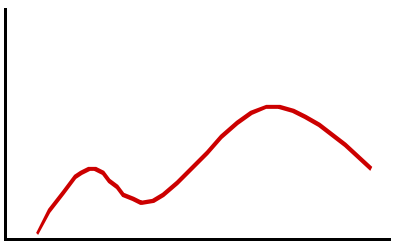
전기적인 신호를 처리하는 방법에는 크게 두 가지가 있습니다. 그 첫번째 방법은 **Analog** 신호 처리이고 다른 한 가지는 **Digital** 신호 처리 입니다. 여러분은 이 두 방법 중 어느 것이 좋다고 생각하십니까? 요즘 Digital이란 문구가 여러 매체에 자주 등장하고 또 아주 중요한 개념처럼 여겨지고 있는데 혹시 Analog보다 Digital이 더 우수한 것이라 생각하세요? 결론을 말하자면 두 가지 방법 중 어느 것이 더 좋은 것은 없습니다. 서로의 장단점이 너무나 분명하기 때문이죠. 다시 말해서 Analog와 Digital 신호 처리의 병행이 가장 이상적인 (혹은 반대로 가장 안 좋아 질 수도…)방법 입니다.

하지만 논리 회로 있어서는 그 처리 신호로 Analog 신호 보다 Digital 신호가 훨씬 우수하다고 할 수 있습니다. 그 이유를 알기 위해서는 Analog와 Digital이 무엇인지 알아야겠죠? 그러면 우선 Analog에 대해서 알아 봅시다.

어떠한 수치가 끊김이 없는 연속적인 변화를 보일 때 이를 **Analog**라 부릅니다. 여기서 중요한 것은 '연속' 이라는 단어입니다. 어떠한 수치가 연속적이라는 것은 두 개 이상의 수치 사이에 중간 값이 존재한다는 것을 의미합니다. 이게 무슨 말일까요?(물론 다들 알고 계시겠지만…) 길이를 재는 Analog도구인 '자'를 생각해 봅시다. 자의 두 지점 A, B를 정하면, C=(A+B)/2 라는 값이 A와 B사이에 존재합니다. A와 C사이에는 D=(A+C)/2 라는 값이 존재하고, A와 D 사이에는……

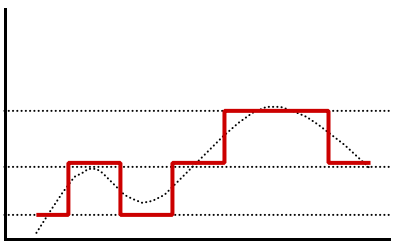
즉, 임의의 두 지점 사이에는 무한히 많은 또 다른 값이 존재하며, 이를 '연속적이다' 라고 말하는 것이죠. 시간, 거리, 기온 등 자연계의 양은 거의 모두 Analog적인 경향을 보입니다. 우리가 주변에서 보는 것 뿐만 아니라 빛, 소리 또한 Analog라 할 수 있죠.

< analog >



Continuous time

< digital >



Continuous time

그러면 **Digital**은 무엇일까요? 유한개의 불연속적이고 분산된 symbol들(digits)로 이루어진 수치 또는 data를 digital information이라 부릅니다. 10개의digit(0,1,2,3,4,5,6,7,8,9)로 이루어진 수 체계(number system)가 가장 좋은 예 이지요. 여기서 중요한 단어는 '**불연속**', '**분산**' 입니다. 연속되는 Analog data와는 달리 Digital data는 중간 값이 존재하지 않습니다. data를 구성하는 digit의 종류와 그 개수가 유한하기 때문에 analog적인 자연계의 양을 무한히 정확하게 표현할 수 없는 것이죠. 즉, 3개의 10단위 digit(ex. 10진수)로 1.99와 2.00을 표현할 수는 있지만, A=(1.99 + 2.00)/2 라는 data를 표현 할 수는 없습니다.

또한, 자연계가 analog적인 것과는 달리 인간의 정보 전달 과정은 digital적인 성격이 강합니다. 인간의 언어가 문자(한글 자모, alphabet,... etc.)라는 유한 개 digit로 구성된 data임을 봐도 알 수 있죠.

2.1.2. Why Digital? – Digital의 장점

Digital system의 중요한 장점은 탁월한 **error correction** 능력입니다. Digital system의 output은 불연속적인 특성을 갖기 때문에 input data의 사소한 변화(error or variation)는 output에서 정확한 하나의 data로 출력됩니다. 이와는 달리, Analog system에서는 input data의 error가 output에 연속적인 값으로 전달되므로, 이는 복잡한 circuit내부의 data처리 과정에서 몇 단계의 과정을 거치면서 error가 쌓이고 증폭되어 최종 output에 예측할 수 없는 변화를 가져오게 됩니다.

Digital system의 또 다른 장점은 정확도를 증가 시키기가 쉽다는 것입니다. 가장 흔히 드는 예시 중 하나인 시계를 살펴봅시다. analog시계에서 시침의 정확도를 높이기 위해선 분침이라는 새로운 system을 추가해야 합니다. 분침의 정확도를 높이려면 초침 system을 추가해야 하며, 더 나아가 ms(milli sec), ns(nano sec)로 정확도를 높이려면 도대체 어디다 어떻게 무엇을 더 달아야 하는지조차 상상하기 힘들게 됩니다. 이와는 달리 digital system에서는 단순히 system clock을 세분하는 logic을 추가하여 이를 표현하는 몇 개의 digit를 더 사용하기만 하면 되는 것입니다. 물론 더 추가하는 것이 말처럼 쉽지는 않겠지만, analog system의 경우보다는 훨씬 간단한 것은 사실이지요.

Digital system은 정보 처리 과정에서 막강한 능력을 발휘합니다. Digital system의 가장 간단한 형태인 binary system은 현대 hardware system의 근간을 이루고 있으며, 증기기관의 발명에 버금가는 인류사의 변혁을 주도하고 있다고 해도 과언이 아닙니다. 단 두개의 digit로 이루어져있으므로 물리적인 양(voltage difference, magnetic polarization or the flow or lack of electrical current)으로 간단히 encode할 수 있으며, 각각 1과 0이라는 symbol을 사용하여 "참", "거짓" 또는, "on", "off" 를 지칭하도록 logical operation에 쉽게 적용할 수 있습니다. 즉, **digital system**은 정확하고 안정적이며 제작이 용이한 정보처리 방식입니다. 따라서 위와 같은 이유에 의해서 논리회로에서 Digital 방식이 더 우수하다고 할 수 있는거죠.

그러나, 위를 이유로 analog system의 장점을 간과할 수는 없습니다. 실제로, Analog

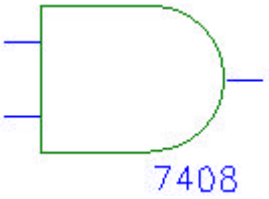
system과 Digital system사이에 우열은 존재하지 않으며, 두 system은 상호 보완적인 관계에 있습니다. Real world는 analog system이며, digital system은 logical world에 존재하므로, real world의 analog data는 logical world의 digital system으로 입력 되어야 하며, digital system에 의해 처리된 정보는 real world로 튀어 나와야 사람에게 쓸모가 있는 analog data로 환원될 수 있습니다. 현재 가장 각광을 받고있는 무선 데이터 통신의 전달 매체인 전파 역시 analog system이므로, 정보의 전달 과정에서도 중요한 역할을 하고 있습니다. 즉, digital system은 정보의 연산과 처리를 담당하며, analog system은 주변 환경을 감지하고 조작하며 data를 외부에 출력하고 전달하는 interfacing circuits로써의 역할을 담당하게 됩니다.

2.2. 논리 회로

그럼 circuit에서 Digital 신호(0과 1)는 어떻게 나타나고 인식될까요? 회로에서 Digital 신호는 Voltage로써 표현 되며 일반적인 회로에서는 5V가 1을, 0V가 0을 나타냅니다.

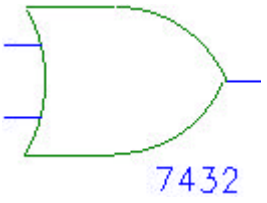
2.2.1. Gates & Truth Tables

- **AND** gate: 두 수가 모두 TRUE(1)일 때만 결과가 TRUE(1)



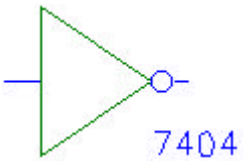
C	D	C AND D
0	0	0
0	1	0
1	0	0
1	1	1

- **OR** gate: 두 수 중 어느 하나라도 TRUE(1)이면 결과는 TRUE(1)



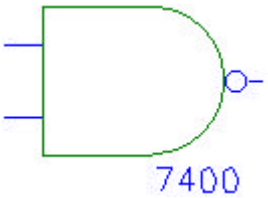
C	D	C OR D
0	0	0
0	1	1
1	0	1
1	1	1

• **NOT** gate: 피연산자의 값을 바꾼다(TRUE(1) -> FALSE(0), FALSE(0) -> TRUE(1))



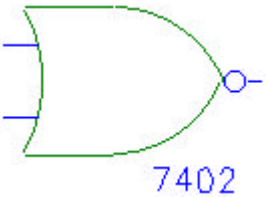
C	NOT C
0	1
1	0

• **NAND** gate: AND한 결과를 NOT하는 연산



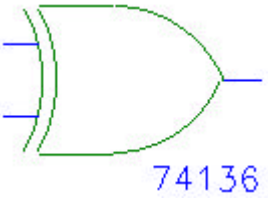
C	D	C NAND D
0	0	1
0	1	1
1	0	1
1	1	0

• **NOR** gate: OR한 결과를 NOT하는 연산



C	D	C NOR D
0	0	1
0	1	0
1	0	0
1	1	0

• **XOR** gate: 두 수가 서로 다를 때 결과가 TRUE(1)인 연산, Exclusive OR



C	D	C XOR D
0	0	0
0	1	1
1	0	1
1	1	0

2.2.2. Application

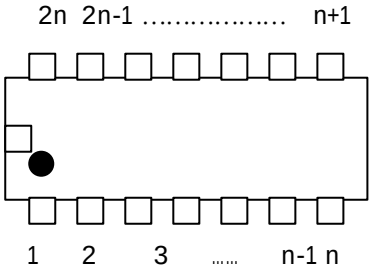
예제) '*' 연산의 Truth Table이 다음과 같을 때 '*' 연산의 논리회로는 어떻게 될까요?

C	D	C * D
0	0	1
0	1	0
1	0	1
1	1	1

3. Basics of Circuit

3.1. CHIP의 핀 배치 및 번호

아래 그림은 CHIP의 핀 배치를 개략적으로 그려 놓은 것이다.



• CHIP을 위에서 보았을 때, 그림과 같이 원형으로 움푹 패인 표시가 있(거나 검은색 점이 있)는 곳을 왼쪽으로 하면 왼쪽의 첫번째 핀이 1번 핀이 된다. 그리고 번호는 1번 핀에서부터 시계 반대 방향으로 차례로 붙여진다.

♦주의 사항♦

실제로 보드에 납땜을 할 때는 기판을 뒤집어서 하게되므로 핀번호 배치가 바뀌게 된다. 그리고 소켓을 사용하지 않고 바로 CHIP을 보드에 납땜하는 경우에는 과다한 열이 가해지지 않도록 한다. 과다한 열이 가해지면 칩이 탈 수가 있다.

• 보통의 경우 TTL계열의 CHIP은 중간번째 핀이 GND이고 마지막 핀이 Vcc인 경우가 많다.(그림에서 n: GND, N: Vcc) 그러나 칩마다 Vcc 와 GND 는 항상 확인하고 납땜을 해야 한다.

3.2. Resistor의 값 읽는 법

A B C D

• 저항(resistor)의 값은 저항에 그려진 줄의 색깔을 통해 알 수 있다. 우선 금색(또는 은색) 줄을 마지막 줄로 보고 그곳이 뒤쪽 이므로 앞쪽부터 차례로 색깔을 읽으면 된다.

• 그림에서 A와 B 줄 색깔은 저항값의 유효 숫자를 나타내고 C는 저항값의 지수값(10의 몇 승), D(금색 또는 은색)은 값의 허용 오차를 나타내는 것이다.

•색깔에 따른 값

0	1	2	3	4	5	6	7	8	9
검정	갈색	빨강	주황	노랑	초록	파랑	보라	회색	백색

• 예) 저항에 그어진 줄이 차례로 갈색, 검은색, 빨간색, 금색이면 이 저항값은,

10 * 10^**2** = 1000 = 1KΩ

이 된다.

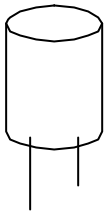
3.3. Capacitor의 값 읽는 법과 극 판별법

• capacitor의 값은 몸통에 써 있는 숫자를 통해 알 수 있다. 기본 단위는 pF(Pico Farad) 이며 숫자중 마지막 숫자는 10의 몇 승인지를 의미한다. 따라서 만일 capacitor에 223이라고 쓰여 있으면,

22 * 10^**3** = 22000pF = 22 nF (Nano Farad)

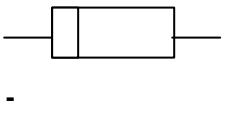
이란 뜻이다. 보통의 패럿단위는 μF (Micro Farad) 이다.

• 극성은 다리의 길이나 표시로 알 수 있는데, 다리가 짧거나 혹은 '-'라고 쓰여진 쪽이 –극이다.(값이 매우 작은 경우에는 극성이 없는 것도 있다.)

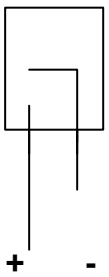


Diode, LED 극 판별하는 법:

• diode의 경우에는 몸통에 줄이 그어진 쪽이 –극이다.



• LED(Light Emitting Diode)의 경우에는 capacitor와 마찬가지로 다리가 짧은 쪽이 –극이며, 긴 쪽이 +극이다. 그리고 LED내부를 봤을 때 'ㄱ'자로 꺾여진 쪽이 –극이다.



3.4. 납땜하는 법

1) 인두를 달군다. (코드를 꼽고 조금만 기다리면 알아서 달궈진다.)
2) 인두에 납을 약간 묻혀서 달궈지는걸 촉진한다. (어느 정도 달궈진 후에 납을 약간 묻힌다고 생각하면 된다.)
3) 먼저 핀 아래쪽에 인두를 대서 핀과 그 밑의 원형 쇠를 달군다. (납땜을 할 부분을 생각하고 그 반대쪽 편을 달군다)
4) 달궈진 핀에 인두를 댄 상태로 납을 녹인다. 키세스 초콜릿 모양이 되게 만들어야 함을 염두에 두고 납을 적당량 녹인다.
5) 인두를 떼고 납을 뗀다. 키세스 모양이 되었음을 확인한다.
6) 동그란 모양이 된 납땜을 냉납이라고 하며, 이는 접촉 불량 등의 원인이 될 수 있으므로 주의한다.
7) 플러그를 뺀다.

3.5. 기타 각종 공구 및 장비 사용법

- 인두(solder)
- 스트립퍼(stripper)
- 오실로스코프(oscilloscope)
- 멀티미터
- Etc.

인두(solder) : 인두는 적혀있는 소모 전력에 따라 파워가 결정된다. 빨리 달궈지는 것이 있는가 하면 달궈지는데 시간이 좀 걸리는 것이 있다. 20W 정도가 가장 적당하다고 생각한다.

스트리퍼(stripper) : 전선(wire) 의 피복을 벗기는데 사용하는 기구이다.

오실로스코프(oscilloscope) : GND(0 V)에 비한 상대적인 전압의 크기를 그림으로 보여주는 장치로 한 화면에 파형을 볼 수 있기 때문에 어떤 파형이 나오고 있는지 확인할 수 있다. 오실로 스코프는 고가의 장비이므로 사용할 때 조심하자.

멀티미터(multi-meter) : 오실로스코프가 파형의 크기를 그래프로 보여준다고 하면, 멀티미터는 숫자로 보여주는 기계이다. 오실로스코프는 시간에 따라 나오는 파형의 모양을 중심으로 생각하는 기계이고, 멀티미터는 정적(time independent)으로 저항의 크기나 전압 전류의 크기 등을 재는 기계라고 생각하면 된다.

¨ **Etc**

- 라디오 펜치 : 코(?)가 긴 펜치이다. 이것저것 구부리거나 자를 때 사용한다.
- 납 흡입기 : 납땜을 잘못했을 때, 흡입기를 사용하여 납을 빨아들인다. 납을 녹인곳에 흡입기를 갖다대고 빨아들이면 납이 쏙 빨려 올라간다. 마방 사람들은 이것을 그냥 뽁뽁이 등으로 부르고 있다.

3.6. 기판에 납땜할 때의 주의사항

1) Vcc 와 GND 를 먼저 납땜한다. (이건 반드시 지켜야 할 사항이다.) 나중에 데이터 어드레스 선을 연결하고 나서 Vcc GND 를 납땜하려 하면 기판이 너무 복잡해 진다.

2) 기타 잡선을 납땜한다.

3) 어드레스선과 데이터 선을 납땜한다.

♣ 이 순서는 반드시 지켜야 하는 것은 아니지만 이렇게 하는 것이 기판을 좀더 깨끗하게 만드는 방법중에 하나이다.

4. Micro Processor and Controller

4.1. Micro Computer

4.1.1. Stored Programming

2차 세계 대전 중에 포탄의 탄도 계산을 위해 개발된 컴퓨터는 그 후 비약적인 발전을 하였습니다. 그러나 처음부터 지금과 같은 모습은 아니었죠. 처음의 컴퓨터들은 수행 동작을 바꾸기 위해서 핀 보드로 직접 회로의 배선을 바꾸는 방식이었습니다. 따라서 한 작업을 수행하는 것에는 문제가 없었지만 다른 작업을 컴퓨터로 하려고 하면 매우 힘들었습니다. 효율성도 그만큼 낮았겠죠. 따라서 이러한 문제점을 개선하기 Stored Program 방식이 고안되었습니다.

Stored Program 방식이란 컴퓨터가 수행해야 할 동작을 한 스텝식 명령을 나열하여 표현하고 이를 내부에 기억시킨 후에 순차적으로 그 명령에 따라 동작하는 방식을 말합니다. 여기서 '프로그램'이라는 개념이 생기게 되었고 컴퓨터를 소프트웨어와 하드웨어로 구분지울 수도 있게 되었습니다.

4.1.2. Micro Computer

Micro Computer란 컴퓨터의 주요 기능(컴퓨터의 3요소- 연산, 기억, 입출력)을 몇 개의 칩(LSI, VLSI)에 내장한 것을 말합니다. 시간이 지날수록 컴퓨터의 복잡한 계산능력보다는 간편한 Controller로써의 기능에 대한 수요가 늘면서 일반화 되었죠. Micro Computer도 일종의 Computer이기 때문에 정해진 프로그램에 따라 동작을 수행하는 것에는 변함이 없습니다. 단지 그 기능이 PC에 비해서 축소되고 특정 용도에 맞게 변화되었다는 차이점이 있을 뿐입니다.

4.2. Micro Processor

4.2.1. CPU – Central Processing Unit

사람으로 치면 두뇌에 해당하는 부분이라고 할 수 있습니다. Central Processing Unit이란 말에서 보듯이 중앙에서 시스템의 제어를 맡고 있는 부분입니다. 컴퓨터의 3요소 중 연산 기능을 담당하죠.

위에서 살펴보았듯이 Stored Program 방식에서는 실행해야 할 동작을 1스텝씩 명령으로

표현하고 기억시켜 그것을 순차적으로 실행함으로써 전체의 동작을 제어하게 됩니다. 따라서 CPU는 다음과 같은 구성으로 이루어져 있습니다.

ALU – Arithmetic Logic Unit

ALU는 산술 논리 연산 장치로 CPU내부에서 수치 계산과 논리 연산이 이루어지는 곳입니다. 보통 Accumulator라 불리는 레지스터에 수행될 데이터가 저장된 후 이를 가지고 연산을 하게 되죠.

CU – Control Unit

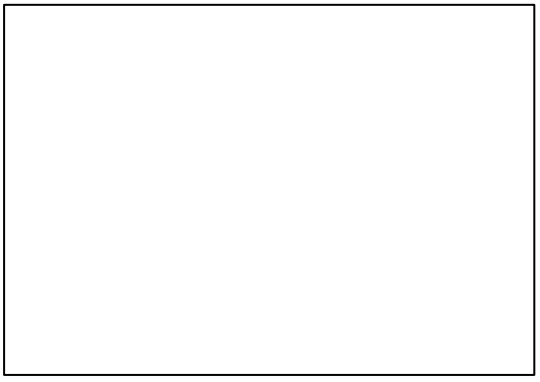
CU는 말 그대로 CPU 내부 동작의 제어를 담당하는 부분입니다. 메모리에서 데이터를 읽어들이거나(fetch) 다음 차례에 수행될 명령어를 부르고 또 Instruction Decoder에게 해석을 지시하는 등의 역할을 수행합니다.

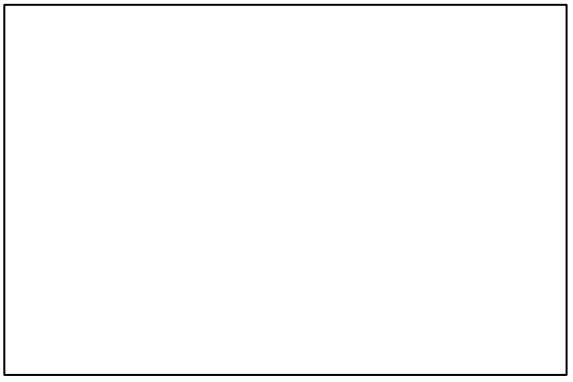
Register

컴퓨터 내에서 명령어나 데이터를 유지하기 위한 작은 저장 공간을 레지스터라고 합니다. 메모리에 저장되어 있는 데이터가 바로 연산에 이용되는 것이 아니라 레지스터에 내용이 저장된 후 이 레지스터(범용 레지스터)들이 실제 연산에 이용됩니다. 레지스터는 연산에만 이용되는 것이 아니라 동작하는데 필요한 여러 정보를 저장하기도 합니다. PC(Program Counter)라는 레지스터는 다음에 수행될 명령어의 위치를 저장하고 있으며, Accumulator는 연산에 쓰일 데이터를, SP는 스택의 포인터를 저장하고 있죠.

Bus

컴퓨터는 하드웨어적으로 전기 회로입니다. 회로와 회로 사이에는 배선으로 연결되어 있고 데이터 교환은 전시 신호를 넘겨줌으로써 실현되게 되죠. 만일 모든 회로를 개별로 배선하면 방대한 규모가 되기 쉽습니다. 따라서 공용 배선을 사용하는 버스 방식이 고안되었죠. 회로에서 버스란 쉽게 말하면 '의미 있는' 전선의 한 묶음 단위라고 할 수 있습니다. 버스에는 크게 Address, Data, Control 버스가 있습니다. CPU내에서도 이러한 버스를 통해 데이터가 이동하며 외부 회로에서도 마찬가지입니다.





4.2.2. Micro Processor

Micro Computer 중에서 CPU부분만을 칩(LSI, VLSI)으로 만든 것이 있는데 이것을 Micro Processor 혹은 MPU(Micro Processing Unit)이라고 합니다. 따라서 시스템 내에서 Micro Processor는 각 신호들을 제어하고 동작을 수행시키며 산술 연산을 비롯한 여러 연산을 수행하는 역할을 합니다.

4.3. Other Parts

위에서 설명한 것과 같이 Micro Processor는 단지 제어와 연산기능을 담당할 뿐입니다.
따라서 Micro Processor만으로는 원하는 동작을 수행하는 시스템을 만들 수가 없습니다. 즉 Micro Processor의 제어에 따라 역할을 수행하는 다른 부분이 필요하죠.
우선 사용하는 데이터를 저장하는 공간(메모리)이 필요하고, 이를 외부와 입,출력할 수 있게 해주는 부분(포트)도 있어야 합니다. 또 일정한 시간을 측정해주는 부분(타이머)도 있어야 하겠죠. 그리고 물론 각 시스템마다 필요한 특수기능을 수행하는 부분도 있어야 할 것입니다. 그러면 각각에 대해서 살펴보도록 하겠습니다.

4.3.1. Memory

메모리는 사용하는 데이터를 저장하는 공간입니다. 컴퓨터의 3요소 중 기억 기능을 담당하는 부분이죠. 흔히 PC에서의 주기억장치(RAM)를 칭하기도 합니다. 그러나 넓은 의미로는 주기억장치뿐만 아니라 레지스터, 캐쉬, 보조기억장치(HDD, Type 등)를 포함합니다.
메모리는 저장되는 방식에 따라 크게 두 가지로 나눌 수 있습니다. 특별한 방식으로 데이터가 저장되기 때문에 중간에 데이터를 바꿀 수는 없지만 전원이 공급되지 않아도 데이터가 보존되는 메모리와 순간순간 프로그램으로 저장된 데이터를 바꿀 수는 있지만 전원이 끊기면 모든 데이터가 사라지는 메모리가 그 두 가지입니다. 전자를 ROM이라 하고 후자를 RAM이라 합니다. 물론 요즘에는 FLASH라고 해서 두 가지 메모리의 장점을 모두 지닌 메모리도 있습니다.

ROM - Read Only Memory
'Read Only Memory'라는 말 뜻처럼 ROM은 읽을 수만 있는 메모리입니다. 따라서 데이터를 ROM에 쓰기 위해서는 ROM Writer가 필요하죠. 보통 Writer를 이용해서 ROM에 프로그램 데이터를 저장시키는 것을 '프로그램을 굽는다'라고 표현합니다.
ROM의 종류에는 크게 4가지가 있습니다.

Mask ROM

공장에서 미리 쓰여져 나온 것으로 지울 수가 없습니다.

PROM(Programmable ROM)

ROM Writer를 이용해서 프로그램을 구울 수는 있지만, 다시 지울 수는 없습니다.

EPROM(Erasable PROM)

자외선으로 데이터를 지울 수 있는 PROM입니다. 칩으로는 27C256이 있습니다.

EEPROM(Electrically EPROM)

자외선이 아닌 전기 신호로 저장되어 있는 데이터를 지울 수 있는 ROM으로 소형 시스템에서 가장 널리 쓰입니다. 대표적인 칩으로 29C256이 있습니다.

RAM – Random Access Memory

'Random Access Memory'란 뜻에서 알 수 있듯이 수시로 저장된 내용을 지우고 다시 쓸 수 있는 메모리입니다.

RAM은 크게 2가지로 나눌 수 있습니다.

SRAM(Static RAM)

전원이 유지되는 한 데이터를 보존하는 RAM으로 제어가 간단하고 고속입니다. 주로 소용량 기억 장치 구성에 쓰입니다. 대표적인 칩으로 62256(혹은 61256, 61256은 62256의 narrow type)이 있습니다.

DRAM(Dynamic RAM)

일정 시간마다 데이터를 재생시켜주는 재생(refresh)회로가 필요한 RAM입니다. 주로 대용량 기억 장치 구성에 쓰입니다.

4.3.2. I/O Port

I/O Port란 Input, Output Port의 약자로 시스템과 외부를 연결시켜주는 역할을 하는 부분입니다. 외부로부터 입력을 받거나 혹은 외부로 출력을 할 때 쓰이며 내부의 다른 디바이스를 제어할 때도 이용됩니다. 여기서 내부, 외부의 경계는 디바이스 레벨에서 생각할 것인지, 칩(LSI or VLSI) 레벨에서 생각할 것인지에 따라 달라지게 됩니다. 디바이스 레벨에서 본다면 I/O Port는 입출력장치를 가리키고 칩 레벨에서는 CPU와 다른 입출력장치 사이의 회로를 가리킵니다. 보통 I/O Port라고 하면 디바이스 레벨의 Port를 말합니다.

I/O Port는 통신 방식에 따라서 크게 직렬과 병렬 두 가지로 구분됩니다.

1) 직렬 포트(Serial Port)

데이터 핀에 들어온 병렬 데이터를 직렬 데이터로 바꾸어서 출력하거나 반대로 직렬 데이터를 병렬 데이터로 바꾸어주는 포트입니다. 컴퓨터와 직렬 통신을 할 때 주로 사용되며 대

표적인 칩으로 8251A가 있습니다.

2) 병렬 포트(Parallel Port)

데이터를 병렬로 입, 출력하는 포트입니다. 다른 I/O 디바이스에서 입력을 받거나 디바이스에 출력을 할 때 주로 사용되며, 대표적인 칩으로는 8255A가 있습니다.

4.3.3. Timer

타이머(Timer)는 clock 수를 세주는 디바이스로 거의 모든 시스템에 이용되고 있습니다. 물론 프로세서가 이러한 기능을 수행할 수도 있지만 이럴 경우 정확성을 보장하기 힘들 뿐만 아니라 대개의 경우 여러 타이머 소스-여러 개의 타이머-가 필요하기 때문에 타이머 디바이스를 쓰게 됩니다. 간단히 동작을 설명하면 프로세서가 타이머에게 몇 clock을 세라 혹은 몇 초를 세라고 명령하면 타이머는 그 시간이 지난 후 프로세서에게 알려주는 방식입니다. 대표적인 칩으로는 8253이 있습니다.

4.3.4. A/D and D/A Converter

A/D(Analog to Digital), D/A(Digital to Analog) Converter는 말 그대로 아날로그 신호를 디지털 신호로 바꿔주거나 반대로 디지털 신호를 아날로그 신호로 바꿔주는 디바이스입니다. 음성 신호나 영상 신호 같은 아날로그 신호를 A/D Converter를 통하여 디지털 신호로 변환함으로써 메모리에 저장할 수 있고 또 D/A Converter를 통해 이렇게 저장된 신호를 복원할 수도 있습니다.

4.4. Micro Controller

4.4.1. One Chip Solution

집적 기술이 떨어졌던 옛날에는 각각 Micro Processor, Memory, I/O Port. Timer 등의 기능을 하는 칩으로 회로를 구성하였습니다. 그러나 집적 기술이 발전하고 또 거의 대부분의 소형 시스템에서 위에서 언급한 기능들은 필수적인 부분이었기 때문에 각 기능을 합치려는 시도가 일어났습니다. 한 칩 안에 Micro Processor, Memory, I/O Port, Timer 등을 내장함으로써 소형화할 수 있을 뿐만 아니라 처리 속도도 높일 수가 있게 되었죠. 이것을 가리켜 One Chip Solution이라고 합니다. 말 그대로 하나의 칩으로 원하는 기능을 해낸다는 뜻입니다.

4.4.2. Micro Controller

Micro Controller는 이러한 One Chip Solution의 한 형태라고 할 수 있습니다. 흔히 MICOM이라고 부르는 Micro Computer의 기본적인 기능을 하나의 칩에 내장한 것입니다.
따라서 대부분의 Micro Controller안에는 Micro Processor, I/O Port, Timer, 그리고 소형의 Memory 등이 있습니다. Memory의 경우에는 그 칩의 크기가 용량에 비례하기 때문에, 내부에 작은 Memory가 있더라도 외부에 더 큰 용량의 Memory 칩을 따로 다는 것이 보통입니다.

4.5. Address and Data

4.5.1. Address

컴퓨터에서의 어드레스란 데이터가 저장되어 있는 위치를 의미합니다. 당연히 데이터는 메모리에 저장되어 있겠죠. 메모리에는 각 데이터마다 주소가 정해져 있습니다. 따라서 어떤 위치에 저장되어 있는 데이터가 필요하다면 그 주소를 알아내어 읽어 들이면 되겠죠. 앞에서도 말했듯이 하드웨어인 컴퓨터는 기본적으로 0과 1일 전기적 신호만을 구별할 수 있습니다. Controller와 메모리 사이에는 Address Bus가 있고 이를 통해서 메모리에게 원하는 주소를 알릴 수 있습니다. 따라서 Address Bus의 bit수에 따라 접근할 수 있는 메모리의 용량이 정해지게 됩니다. 즉 Address Bus의 bit수가 n이라면 2의 n제곱에 해당하는 영역의 메모리를 접근할 수 있죠.
예를 들어서, 어드레스가 16bit인 프로세서라면 2의 16승, 65536개의 주소 영역을 가지고 있는 것입니다. 2진수로 생각한다면 16bit의 묶음은 0000000000000000부터 1111111111111111까지 표현할 수 있습니다. 따라서 2의 16승이 되는 것이죠. 그리고 만일 메모리에 저장되는 데이터의 한 단위가 한 바이트(byte)라면 65536byte, 64Kbyte의 메모리를 쓸 수 있는 것입니다.
그러나 어드레스가 메모리 접근에만 이용되는 것은 아닙니다. 앞에서 언급한 I/O Port같은 다른 부분들을 사용할 때도 이용할 수 있습니다. 이를 Memory Mapping이라고 하죠. I/O Port같은 디바이스의 Control신호에 어드레스를 물려 제어하는 것입니다. 이렇게 하면 많은 디바이스를 연결하여 쓸 수 있습니다.

4.5.2. Data

데이터란 말 그대로 컴퓨터 내에서의 자료를 의미합니다. 일반적인 의미의 데이터 뿐만 아니라 명령어도 데이터의 한 종류로 보기도 하죠. 흔히 몇 비트 머신이라는 말을 하는데 여기서 몇 비트에 해당하는 것이 데이터입니다. 즉 32비트 컴퓨터는 내부적으로 32비트(4바이트)의 데이터가 한 단위로 처리됨을 의미하는 것 입니다.

예를 들어 Z80이란 프로세서는 어드레스는 총 16bit(A0~A15)이고 데이터는 8bit(D0~D7)인 8bit 프로세서입니다. 어드레스가 16bit이므로 위에서 말한 것과 같이 총 64K개(정확히는 65536개)의 주소를 가질 수가 있죠. 그리고 데이터 비트가 8bit이므로 64K * 8bit, 즉 64Kbyte의 메모리 영역을 가지고 연산을 수행하게 됩니다. 프로세서 내부의 레지스터 등도 모두 8bit이기 때문에 8bit을 한 단위로 처리할 수 있습니다.

4.6. Memory Decode and I/O Decode

4.6.1. Memory Decode

보통의 Micro Controller의 경우 동작을 하기 위해서는 ROM과 RAM이 둘 다 필요합니다. 따라서 두 메모리 칩을 다 사용하기 위해서는 controller에서 접근할 수 있는 메모리 영역을 각각에게 할당해야 하죠. 이 때 어드레스의 특정 bit를 이용하여 영역을 나누어 구분하는 것을 Memory Decode라고 합니다.

예를 들어 어드레스가 16bit이라면 총 64K의 주소를 가질 수가 있는데 이중 상위, 즉 0부터 32K-1번째까지를 ROM에 할당하고 나머지를 RAM에 할당하려고 한다면 어드레스 15번 A15를 ROM에 CE(Chip Enable, active low)에 연결하고 이 것을 NOT gate를 거쳐 RAM의 CE에 연결하면 됩니다. CE가 active low이므로 A15가 0일 때, 즉 번지가 0000에서 7FFFF까지는 ROM이 enable되고, 반대로 A15가 1이 되면 RAM이 enable되므로 각각에 어드레스가 할당되었음을 알 수 있습니다.

4.6.2. I/O Decode

Decoder 등을 이용하여 많은 I/O Port를 제어하는 것을 말합니다.

4.7. Operation of Micro Controller

Micro Controller의 간단한 수행과정을 알아보도록 하겠습니다. 우선 Micro Controller의 어드레스와 데이터가 모두 16bit이고, 외부 메모리로 각각 32Kbyte의 용량을 지닌 ROM과 RAM을 버스를 통해 연결하였다고 가정하도록 하죠. ROM과 RAM은 Memory Decode의 방법으로 어드레스 15핀(A15)을 이용하여 구분하였습니다. 따라서 ROM에는 상위 번지(0000~7FFFF)를 할당하였고, RAM에는 하위번지(8000~FFFF)를 할당할 수 있습니다.

보드에 전원을 넣게 되면 controller가 reset(power-on reset)이 되고, 프로그램 카운터 PC가 0000번지를 가리키게 됩니다(실제와는 다를 수 있음). 따라서 0000번지에 있는 명령어부터 차례로 프로그램이 수행되게 됩니다. ROM에는 미리 저장시켜 둔 프로그램(예를 들어 모니터 프로그램)이 들어있겠죠. 하위 영역을 RAM으로 지정함으로써 프로그램 수행 시 필

요한 스택영역이나 임시 데이터를 저장할 수 있습니다. 또한 자신이 만든 프로그램을 동적으로 저장하여 실행 시키고자 할 때도 RAM을 이용하면 됩니다. 외부와의 입출력을 담당하는 포트(e.g. 시리어 포트)를 이용하여 RAM에 자신의 프로그램을 저장시키고 ROM에 저장되어 있는 프로그램으로 하여금 PC를 RAM의 첫 번지(8000)에 위치하도록 만드는 것 입니다. 그렇게 되면 RAM에 저장되어 있는 프로그램이 실행되게 되죠.

매번 테스트할 때마다 ROM에 프로그램을 구워서 실행시키는 것은 힘이 들죠. 따라서 보통 ROM에는 모니터 프로그램과 같은-시스템을 초기화 해주고 포트를 통해 프로그램을 전송받아 RAM에 저장시키는 등의 역할을 수행하는-프로그램을 넣고 포트를 통해 RAM에 프로그램을 다운로드 받아 프로그램을 테스트하게 됩니다.

저장되는 프로그램의 내용에 따라서 I/O Port 등에 특정한 데이터를 내보낼 수도 있고, 타이머를 이용하여 작업을 수행할 수도 있습니다.

5. 80C196KC

5.1. 기초 용어

♦ 컴퓨터는 중앙 처리 장치(cpu), 메모리, 입출력 장치로 구성되는 데, 이 중 cpu를 하나의 집적회로(Integrated Circuit)로 만든 것을 마이크로 프로세서라고 한다.

♦ 마이크로 프로세서는 프로그램한 명령어를 수행하는 제어기, 데이터를 잠시 저장하는 레지스터, 결과를 계산하는 산술논리 연산장치(Arithmetic Logic Unit)으로 구성되어 있다.

♦ ALU : 산술연산이란 이진수의 덧셈을, 논리 연산이란 AND,OR,NOT 등의 논리 연산을 말한다. 이는 논리 게이트 들로 구성되어 있다.

♦ 게이트는 신호를 통과 시키고, 끊고, 선택하는 기능만을 할 뿐 신호를 기억하지는 못한다. 따라서 신호를 기억할 수 있는 소자가 필요한데 디지털 회로에서 신호를 기억하는 최소 단위를 플립플롭이라 한다.

♦ 플립플롭은 입력의 변화의 변화가 그대로 출력에 나타나지만 마이크로프로세서는 동작을 실행할때 기준 클럭에 맞추어 동기화시켜 실행한다. 따라서 플립플롭에도 시간 개념을 추가해 클럭 입력이 필요하다.

♦ 래치란 정해진 순간에 클럭이 입력되면 데이터를 검출해서 저장하는 소자로 플립플롭으로 구성된다.

♦ 레지스터란 마이크로프로세서 내부에 있는 래치를 말한다. 특히 ALU의 출력을 저장하는 레지스터를 누산기(어큐뮬레이터)라 하고 명령어를 저장하는 레지스터를 명령 레지스터 라고 한다.

♦ 리셋 : 프로세서가 리셋되면 정해진 내부 장치들을 정해진 값으로 초기화 한 후 리셋 벡터로 설정된 번지부터 프로그램을 실행한다.

♦ 프로그램의 실행과정은 내부 명령 레지스터로 명령을 읽어 들여 오는 과정(fetch)와 이것을 해독하는 과정(decode), 실제로 실행하는 과정(execute)으로 나눌 수 있다.

♦ 인터럽트 : 프로그램을 순차적으로 실행하는 도중에 현재 처리하고 있는 일보다 더 우선순위가 높은 일을 처리하려고 할 경우 프로세서에 인터럽트를 걸어 프로그램의 실행 순서를 바꿀 수 있다.

5.2. 80C196KC의 내부 구조

5.2.1. 메모리 영역

80C196 계열의 메모리 영역은 총 64KB 이며 대부분을 사용자가 메모리로 사용할 수 있다. 그러나 일반적인 용도로 사용할 수 없는 영역이 있는데, 이 영역은 레지스터 파일 영역이나 인터럽트 벡터 그리고 포트 등의 용도로 사용되는 영역으로 0x0000~0x01FF영역과 0x1FFF~207F 영역이다.

5.2.2. 레지스터 화일

주소 0x0000~0x01FF 번지 사이의 512바이트 영역 중에서 아래쪽 256바이트 영역에 내부 레지스터 화일이, 위쪽 256바이트 영역에 추가 내부 램이 존재한다.

♦ 0x0000~0x0017번지까지의 아래 쪽 24바이트는 특별 기능 레지스터라 하며 io 제어 레지스터가 위치한다.

♦ 0x0018~0x00FF번지까지 232바이트를 레지스터 화일 이라고 한다.레지스터 화일은 바이트(8bit) 워드(16bit) 떠블 워드(32bit) 단위로 읽고 쓸 수 있다.

♦ 0x0100~0x01FF번지의 추가 RAM을 수직 창(vertical register windowing)을 이용하여 레지스터 화일로 투영(mapping)시켜 레지스터로 사용할 수 있다.

♦ 스텍 포인터 : 0x0018~0x0019번지는 스텍 포인터 이다. 스텍 포인터는 사용자가 프로그램에서 초기화 해주어야 하며 16bit 이므로 64KB 메모리 어느 곳이든 가리킬 수 있다. 스택 연산을 스택 포인터를 감소 시키므로, 스택 포인터는 스택의 최고 위치 +2 바이트 위로 초기화 해야 한다.

5.2.3. 특별 기능 레지스터

0x0000~0x0017번지까지의 아래 쪽 24바이트는 특별 기능 레지스터라 하며 io 제어 레지스터가 위치하며 공간의 활용도를 높이기 위해 3개의 수평창(Hwindow)을 가지고 있다. 대부분의 주변 장치는 이 것에 의해 제어 된다.

5.2.4. 수직창

수직창은 0x0100~0x01FF번지의 추가 RAM을 레지스터 화일로 투영(mapping)시켜 레지스터로 사용할 수 있게 해 준다.

5.2.5. 내부 ROM과 EPROM

내부 ROM이나 EPROM을 사용할 경우 0x2080~0x5FFF 번지의 영역이 할당되며,

0x2000~0x207F 번지의 영역이 인터럽트 벡터, 주변 장치 서버 , 칩 설정 레지스터, 그리고 보안키가 위치한다.

5.3.외부 메모리 인터페이스

5.3.1. 버스 동작

♦ 80C196의 버스 폭은 8bit / 16bit /혼용 버스 모드 등으로 선택해서 사용할 수 있다. 196은 주소 버스와 데이터 버스가 같은 핀을 공유 하므로 주소 래치 허용 ALE를 이용하여 주소 버스만을 래치한다. 이 분리된 주소 버스를 A0~A15 ,데이터 버스를 D0~D15로 부른다.

♦ 버스 사이클 : 버스 사이클은 clkout 펄스의 2주기로 구성된다. 따라서 버스 동작을 마치기 까지 4개의 모서리가 발생한다. 버스 사이클은 이 clkout의 하강 모서리에서부터 시작되며 첫번째 하강 모서리에서 주소 래치 허용 ALE 신호가 허용되고 주소/데이터 겸용 버스에 주소 신호가 실린다. 다음 상승 모서리에서는 ALE 가 억제된다. 다음 하강 모서리에서는 읽기 신호 RD가 허용되고 버스는 읽기 동작을 하기위한 준비 상태로 대기한다. 쓰기 동작일 경우에는 하강 모서리에서 쓰기 신호 WR이 허용되고, 버스는 유효한 데이터를 제공한다. clkout의 마지막 상승 모서리에서 읽기 동작의 경우 데이터를 래치해 내부로 읽어 드리고 쓰기 동작의 경우 유효한 데이터를 계속 유지한다.

5.3.2. 칩 설정 레지스터 ccr

♦ 칩의 각종 모드들을 설정하는 칩 설정 레지스터 ccr은 cpu가 리셋된뒤 메모리로부터 가져오는 첫번째 바이트 이다. 0x2018번지에 위치하는 ccr은 전력 절약 모드 허용,버스 폭의 선택 등을 제어한다.

5.3.3. BUSWIDTH

♦ 리셋시 칩 설정 레지스터의 버스 폭 설정 ccr.1=1로 설정해서 16비트 버스 모드를 선택하면 외부 버스의 폭을 BUSWIFTH 핀에 의해 필요에 따라 버스 사이클 마다 바꿀 수 있다. ccr.1=1 상태에서 외부 핀 BUSWIDTH=1로 설정하면 버스 폭은 16비트로 설정 되고, BUSWIDTH=0으로 설정 되면 버스 폭은 8비트로 설정된다.

5.4. 인터럽트

5.4.1. 인터럽트

♦ 80C196KC에는 28개의 인터럽트 소스가 있다. 이 인터럽트 소스들이 모여 15개의 일반 인터럽트 벡터를 이루며, 추가로 NMI , TRAP 명령 처리를 위한 3개의 특별 인터럽트 백터가 있다.

♦ 프로그래머는 인터럽트 벡터 테이블을 그에 해당하는 인터럽트 처리 루틴의 시작 번지로 초기화 시켜 놓아야 한다. 사용 하지 않는 인터럽트 벡터들은 모두 오류 처리 루틴으로 점프하도록 하는 것이 좋다. 그러나 C언어로 쓸 경우에는 각각의 인터럽트에 해당하는 함수가 존재하므로 이를 이용하여 프로그램을 작성 한면 된다.

♦ 인터럽트의 제어는 5개의 레지스터에 의해 이루어 진다.

①인터럽트 대기 레지스터 int_pending ② 인터럽트 대기 1 레지스터 ipend1

③인터럽트 거름 레지스터 int_mask ④ 인터럽트 거름 1 레지스터 imask1

⑤프로그램 상태 워드 psw

♦ 인터럽트 대기 레지스터 : 하드웨어는 16개의 인터럽트 중에서 하나가 발생하면 인터럽트 대기 레지스터 int_pending(0x09), 인터럽트 대기 1 레지스터 ipend1(0x12) 중에서 해당하는 비트를 1로 설정한다. 1로 설정된 대기 비트는 인터럽트가 해당 벡터로 점프하면서 지워진다.

♦ 인터럽트 거름 레지스터 : 인터럽트는 인터럽트 거름 레지스터 int_mask와 인터럽트 거름 1 레지스터 imask1의 해당 비트를 1로 설정하거나 0으로 지워 독립적으로 허용하거나 억제 시킬 수 있다.

♦ 전체 인터럽트 억제 : 특별 인터럽트인 NMI,TRAP 명령 , 그리고 제공되지 않는 명령 인터럽트를 제외한 모든 인터럽트는, 프로그램 상태 워드 psw의 전체 인터럽트 허용 I=0으로 지워 억제할 수 있다. 전체 인터럽트 허용 I=1로 설정하면 인터럽트 거름 레지스터에서 1로 설정되어 있는 모든 인터럽트를 사용할 수 있다.C프로그램에서는 enable(); 과 disable(): 함수로서 전체 인터럽트를 제어한다.

5.4.2. 우선 순위

♦ 우선순위 선택기는 대기중이면서 허용되어 있는 모든 인터럽트를 계산해 우선순위가 가장 높은 인터럽트 한 개를 선택한다. 그러면 인터럽트 제어기는 이 선택된 인터럽트의 인터럽트 벡터로 프로그램을 점프 시킨다. 이 인터럽트 벡터에 인터럽트 처리 루틴의 시작 번지를 위치 시켜야 한다,

♦ 우선 순위는 소프트 웨어에서 인터럽트 거름 레지스터를 제어함으로써 변경할 수 있다.

5.4.3. 특별 인터럽트

♦ 특별 인터럽트로 80C196에는 NMI, TRAP명령 ,그리고 정의 되지 않은 명령 처리를 위한 3개의 특별 인터럽트가 있다.

♦ NMI : 거를 수 없는 인터럽트(Nom-Maskable Interrupt)는 우선 순위가 가장 높은 인터럽트이다. NMI 벡터는 0x203E번지에 위치한다. NMI는 거를 수 없는 인터럽트 이나 역시 인터럽트 이므로 다른 인터럽트와 짝을 맞추기 위해 인터럽트 거름 1 레지스터 imask1에 NMI 거름 비트 imask1.7을 가지고 있다. 물론 NMI는 이 비트를 0으로 해 놓아도 거를 수 없다. NMI는 인터럽트의 상승 모서리에서 발생한다.

♦ TRAP명령 : 0x2010번지에 위치하는 벡터. 단일 명령어 인터럽트로서 소프트 웨어 디버거를 ASKEMFEO 이용하며 이 명령은 바로 다음에 명령의 실행이 끝나면 인터럽트를 발생한다.

♦ 80C196에서 정의 되지 않은 명령어를 실행 시키면 0x2012번지의 제공되지 않은 명령어 인터럽트벡터로 점프한다.

5.4.4. 인터럽트 타이밍

♦ 인터럽트는 발생 즉시 인식되는 것은 아니다. 만약 인터럽트가 명령이 끝나는 시점보다 4 상태 시간 앞서서 발생하지 않았다면 이 인터럽트는 다음 명령이 실행된 후에 인식된다. 이것은 명령이 시작되기까지는 몇 개의 상태시간이 소요되기 때문이다.

♦ 인터럽트가 인식되면 인터럽트 대기 비트는 지워지고 해당하는 인터럽트 벡터로 점프하는 CALL 명령어가 우행된다. 이 CALL명령은 현재 실행 중인 명령이 끝난 후에 발생한다.

5.4.5. 인터럽트 요약

♦ 직렬 포트 인터럽트 : 직렬 포트는 3가지 인터럽트 중에 하나를 발생한다. 보내기 인터럽트 transmit_interrupt , 받기 인터럽트 receive_interrupt, 직렬포트 인터럽트 serial_port 가 그것이다.

♦ 타이머 (software_timer ,timer_overflow…..): 타이머를 미리 초기화 시켜 놓으면 그 시간 동안 타이머가 시간을 세다가 그 시간을 넘으면 친터럽트가 들어와서 인터럽트 루틴을 따르게 되고 다시 원상태가 되면 타이머는 다시 시간을 세기 시작한다.

5.5. 펄스 폭 변조 출력

5.5.1. 펄스 폭 변조 출력

♦ 80C196KC에는 3개의 펄스 폭 변조(PWM) 풀력 퍼트가 있다. 이 펄스 폭 변조 출력을 이용해 디지털 값을 애널로그 양으로 변환할 수 있다.

♦ 구성 : 펄스 폭 변조 부는 펄스 폭의 값을 저장하는 PWM 제어 레지스터, 시간을 세는 카운터, 카운터 값과 PWM제어 레지스터 값을 비교하는 비교기, RS 플립플롭으로 구성되어 있다.

♦ 동작원리 : 8비트의 카운터가 매 상태 시간 마다 증가한다.

(시작)카운터의 값이 0이 될 때 PWM의 출력이 1로 올라간다. → (비교)카운터가 PWM 레지스터의 값과 일치할 때 PWM의 출력이 0으로 바뀐다. → (끝)마지막으로 카운터가 넘치면 출력은 다시 1이 된다.

♦ 오동작 방지 : 카운터 값이 PWM 레지스터를 향해 가는 도중 ,PWM 레지스터에 현재 카운터 값보다 작은 새로운 레지스터 값을 써 넣으면 비교기는 이 새로운 값과 일치하는 시점을 찾지 못하고 오동작하게 된다. PWM 레지스터는 이런 오동작을 발생시키는 것을 막기 위해 항상 카운터가 넘치는 시점에서 임시 래치에 저장된 PWM 레지스터의 값을 다시 가져온다.

♦ 주파수 조절 : 80C196KC는 주파수 반감 비트 ioc2.2를 이용햐 PWM의 주파수를 반으로 줄일 수 있다. 이 주파수 반감 비트 ioc2.2는 PWM의 주기를 매 256 또는 512 상태시간으로 조절한다.

♦ 펄스 폭 설정 : PWM 펄스의 폭은 PWM 레지스터 pwm_contrl에 그 값을 써 넣어 조절한다.

♦ 용도 : PWM 출력은 속도나 위치 등의 제어를 위해 PWM 파형이 필요한 전동기 제어 등에 응용 가능

5.5.2. 아날로그 출력

♦ 아날로그 출력 : PWM 출력이나 고속 출력 장치를 사용하면 회로에서 필요한 아날로그 출력을 만들 수 있다. 그러나 이 방법은 모두 펄스의 폭과 주기가 변하는 시각파를 만든다. 따라서 출력으로 부드러운 아날로그 신호가 필요하다면 이 시각파를 필터링 해야 한다.

♦ 버퍼링 : 낮은 전류 영역에서만 제대로 파형이 보장되는 신호에 충분한 전류를 공급하여 신호가 0에서 5볼트 영역 사이로 확실하게 전달되도록 한다.

5.6. 타이머

♦ 타이머는 감소하는 시계로 정해진 시간부터 시간량이 감소 하기 시작해서 0이 되는 시점이을 알게 하는 것이다. 타이머는 이 넘치는 시점에서 타이머 넘침 상태를 표시하고 타이머 넘침 인터럽트를 발생 시킨다.

5.6.1. 타이머1

♦ 타이머1은 16비트의 자유 증가 타이머로 리셋 기능이 없다.

♦ 클럭 소스 : 매 8번 째 상태 시간마다 증가하며, 상태 시간은 외부 클럭 주파수의 1/2이므로 타이머 1의 클럭은 외부클럭 주파수의 1/16이다.

♦ 타이머1은 수평창0의 0x0A번지에 위치한 타이머1 레지스터 timer1을 통해 그 값을 읽을 수 있으며 수평창 15의 0x0A 번지를 통해 값을 써 넣을 수 있다.

♦ 타이머1넘침 상태: 타이머 1의 넘침 상태는 io상태 1 레지스터 ios1.5=1이면 넘침 ios1.5=0이면 안 넘침 상태이다. 타이머1은 넘치는 상태에서 cpu 에 인터럽트를 발생시킬 수 있다.

5.6.2. 타이머2

♦ 타이머2는 타이머, 카운터 겸용으로써 용도는 HSO 장치를 기준 시간으로, 증가/감소 카운터로, 외부 사건 기록용으로, 또는 추가의 카운터 등으로 사용된다.

♦ 클럭 소스 : 타이머는 3가지 클럭 소스를 가진다.

♦ 고속 증가 모드 : 타이머2는 보통 모드에서는 매 8번 째 모서리마다 증가하고,고속 증가 모드시에는 매번 모서리 마다 증가한다.

♦ 리셋 : 타이머2는 4개의 리셋 소스를 가지고 있다.

♦ 타이머2는 수평창0의 0x0C번지에 위치한 타이머1 레지스터 timer1을 통해 그 값을 읽을 수 있으며 수평창 1의 0x0C 번지를 통해 값을 써 넣을 수 있다.

♦ 타이머2넘침 상태: 타이머 2의 넘침 상태는 io상태 1 레지스터 ios1.4=1이면 넘침 ios1.4=0이면 안 넘침 상태이다. 타이머2는 넘치는 시점에서 cpu 에 인터럽트를 발생시킬 수 있다.

5.7. 고속 입력 장치

♦ 고속 입력 장치(hsi : high speed inputs)는 입력 펄스의 폭을 타이머 1로 측정하는 장치이다. '고속'이라 부르는 이유는 cpu에 거의 부담을 주지 않고 하드웨어가 직접 펄스의 폭을 측정하여 FIFO에 저장한후 인터럽트를 발생하기 때문에 펄스를 입력받는 핀을 hsi라 하고 hsi에는 hsi.0~his.3까지 4개의 핀이 있다. 펄스폭을 측정하는 방법에는 매 상승 모서리마다 측정하는 방법, 그리고 매 하강 모서리마다 측정하는 방법, 매 상승/하강 모서리 모두를 측정하는 방법, 그리고 매 8번째 모서리 마다 측정하는 방법이 있다. 이 모서리의 발생을 사건이라고 한다. 어느 핀에서 어떤 사건에서 입렵받을 것인가 하는 his 모드를 his 모드 레지스터에 써넣으면 고속 입력 장치는 해당하는 사건이 발생할 때 마다 사건 발생 시점을 기록한다.

♦ FIFO : hsi 사건이 발생하면 7개 *20(16+4)비트 FIFO는 타이머1 의 값과 16비트와 어느 핀에서 사건이 발생 했는지를 나타내는 4비트를 함께 저장 한다. 만역 입력으로 여러 핀을 사용한다면 hsi 사건을 처리할 경우 소프트 웨어는 어느 핀에서 사건이 발생 했는지를 알기 위해 각 상태 비트를 검사해야 한다. 이 검사를 위해 hsi 저장 레지스터를 읽으려면 8 상태 시간이 필요하다. 따라서 hsi 시간 레지스터 hsi_time은 연속적으로 읽는데 8 상태시간이 걸린다.

♦ hsi_mode : 각 hsi핀은 4개의 동작 모드가 있다. hsi 모드 레지스터 hsi_mode 가 어느 핀에서 어떤 종류의 사건을 기록할 것인지를 정한다.

♦ hsi 인터럽트 : 고속 출력 장치 는 3가지 인터럽트를 발생한다. hsi FIFO 4 인터럽트(FIFO에 4개 이상의 사건이 저장 될 때), hsi FIFO 가득 참 인터럽트(FIFO에 6개 이상의 사건이 저장 될 때), hsi 데이터 이용 가능 인터럽트(FIFO에서 hsi 저장 레지스터로 매번 값이 넘어 갈 때)

5.8. 고속 출력 장치

♦ 고속 출력 장치는 타이머를 사용해 정해진 시접에서 외부로 펄스를 출력하거나 내부 장치에 명령을 내리는 장치이다. '고속'이라 부르는 이유는 cpu에 거의 부담을 주지 않고 하드웨어가 직접 처리하기 때문이다. 고속 출력 장치에서 펄스를 외부로 출력하거나 내부 장치에 명령을 내리는 일을 사건이라고 한다. 사건은 내부 사건과 외부 사건으로 나누어 지는데 , 내부 사건에는 타이머 2 리셋, AD 변환 시작, 소프트 웨어 타이머 인터럽트 발생이 있고, 출력핀 hso로 펄스를 출력하는 일이 외부 사건이다. 출력 핀 his는 hso.0~hso.5 까지 6개가 있다. 어느 시접에서 어떤 사건을 발생 시킬 것인가 하는 것을 hso 명령 레지스터에

써 넣으면 고속 출력 장치가 해당하는 시점에서 사건을 발생 시킨다.

♦ 고속 출력 장치는 정해진 시간에 사건(펄스 또는 인터럽트)을 발생 시킨다. 어느 핀에서 어던 사건이 발생시켤 것인가 하는 사건의 명령은 hso 명령 레지스터 hso_command 에, 사건이 발생할 상대적인 시점은 hso 시간 레지스터 hso_tiame에 기록한다.

♦ 고속 출력 장치는 발생시킬 사건을 한번에 8개 까지 대기 시킬 수 있으며 이 사건들 중에서 어느 사건이든 발생할 때마다 인터럽트를 발생 할 수 있다. 외부 사건 , 즉 6개의 hso 핀 중에서 1개 이상의 변하는 시점에서 고속 출력 인터럽트 high_speed_outputs가 발생하고, 내부 사건인 AD변환의 시작 , 타이머 2의 리셋이나 소프트 웨어 타이머 인터럽트 등의 명령에서 소프트 웨어 인터럽트 software_timer를 발생 시킨다.

♦ 내용 변경 메모리 CAM : hso 장치에서 제어의 핵심은 내용 변경 메모리(CAM : Constant Addessable Memoryt)이다. 매 상태 시간 마다 하나의 CAM 값이 타이머 값과 비교 된다. 따라서 전체 CAM 값을 타이머 값과 비교하는 데는 8상태 시간이 걸린다. 그러므로 hso의 분해능은 8 상태 시간이다. (상태 시간은 클럭 주파수의 1/2 이므로 8 상태 시간은 클럭 주파수의 1/16 이다.).

5.9. AD 변환기

5.9.1. AD 변환기

♦ 80C196은 애널로그 입력을 처리할 수 있는 ad 변환기를 가지고 있다.

♦ 구성 : ad 변환기는 8 채널 멀티 플랙서 , 샘플-홀더, 10비트 연속 근사 ad 변환기로 구성되어 있다.

♦ 분해능: ad 변환기는 변환시 분해능을 8 비트와 10비트 중에서 선택할 수가 있다. Ad 명령 레지스터의 비트수 ad_command.4의 비트수 조정으로 이는 이루어 진다. 8비트 변환하면 분해능은 떨어지나 상대적으로 변환시간이 짧다.

5.9.2. 변환 시간

♦ 총 변환 시간 : 충전 시간 + 변환 시간

♦ 충전 시간(sample time) : 충전 시간은 애널로그 입력이 샘플 콘덴서에 연결되는 시간이다. 충전 시간이 너무 짧으면 샘플링 콘서트를 적절하게 충전하지 못한다.반대로 충전 시간이 너무 길면 그 사이에 외부 입력 신호가 변하고 따라서 오차가 생긴다.

♦ 변환 시간 : 변환 시간은 샘플 콘덴서에 샘플링 된 애널로그 전압을 디지털 값으로 변환하는데 걸리는 시간이다. 변환시간은 비교기가 애널로그 전압을 비교해서 디지털 값으로 변환한 후 안정될 만큼 길어야 하며, 콘덴서가 방전해서 정확도를 잃지 않을 만큼 짧아야 한다.

5.9.3. AD변환 과정

♦ ad 변환기는 ad명령 레지스터의 변환 시작 ad_command.3=1로 설정 함으로써, 또는 hso 명령인 ad 변환 시작 0x0F에 의해 ad 변환기 제어 회로에 변환 시작 신호를 발생 시킨다.

♦ 80C196은 애널로그 전압을 디지털 값으로 변환하기 위해 연속 근사 알고리즘을 사용하다. 연속 근사 변환 방식은 이 기준 전압들을 애널로그 입력 과 비교 계속 바꿔 가며 입력에 가장 가까운 기준 전압을 찾는 방법이다. 찾을 때에는 0에서부터 최대 전압까지 한방향으로 증가시키며 비교하는 것이 아니라 빨리 찾기 위해 이진 탐색 법을 사용한다.

5.10. 직렬 포트

5.10.1. 직렬 포트

♦ 80C196 의 직렬 포트에는 1개의 동기 보드와 3개의 비동기 보드가 있다. 모드 0은 동기 쉬프트 레지스터 모드로 직렬 포트를 통해 io를 확장하기 위해 설계 되었다. 모드1은 표준 8비트 비동기 보드로 보통의 직렬 통신용으로 사용된다. 모드2와 보드 3은 9비트 비동기 보드로 프로세서 사이의 통신에 사용된다.

♦ 비동기 보드는 모두 보내기와 받기를 동시에 할 수 있는 완전 이중 방식이다.

5.10.2. 직렬 통신 모드

♦ 모드 1 : 표준 비동기 통신 모드.모드1 에서 데이터 프레임은 시작비트(1)+ 데이터 비트(8)+ 정지 비트(1)=10비트 로 구성되어 있다.

♦ 쉬프트 클럭 : 보내기와 받기는 각각 독립된 쉬프트 클럭에 의해 제어된다. 보내기 쉬프트 클럭은 보레이트 발생기가 초기화 되었을 때 시작한다. 받기 쉬프트 클럭은 데이터가 1에서 0으로 떨어 질 때 즉 시작 비트를 받았을 때 리셋된다, 따라서 보내기와 받기 클럭은 주파수는 같지만 서로 동기되지 않을 수 있다.

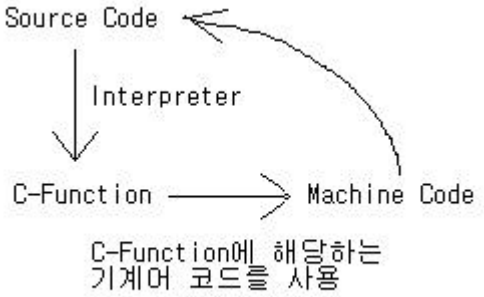
♦ 모드 2 : 비동기 9번 째 비트 인식 모드. 이 모드는 모드 3과 합께 보통 다중 프로세서 통신에 이용된다. 데이터 프레임은 시작비트(1)+ 데이터 비트(8)+ 9번 째 데이터 비트 +정지 비트(1)로 구성되어 있다.

6. C Programming

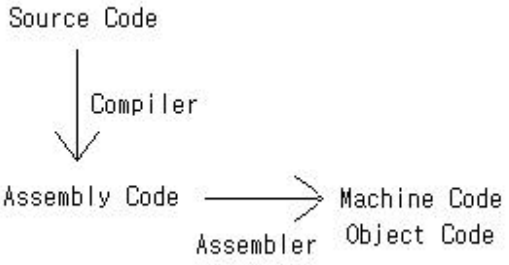
C언어는 코드의 효율성, 구조화 프로그래밍, 호환성이 있는 데이터 구조, 동작 응용 등 범용성이 있는 언어이다. 이번 장에서는 C언어를 개략적으로 살펴보도록 한다.

6.1. Program이란..

'프로그램'이란 단어는 두 가지 뜻으로 사용된다. 프로그래머에 의해 창조된 각각의 명령어 혹은 원시 코드를 말하기도 하고, 또 하나는 실행시킬 수 있는 전체 소프트웨어를 의미하기도 한다. 원시 코드는 두 가지 방법에 의해 실행 프로그램이 될 수 있다. 하나는 인터프리터가 원시 코드를 실행어로 번역해주고, 컴퓨터는 이 실행어를 기반으로 하여 작업을 한다. 또 하나는 컴파일러라는 것이 원시 코드를 실행 프로그램으로 바꾸어주는데, 이는 나중에 언제라도 실행시킬 수 있다. 인터프리터 방식은 작업하기 쉬운 반면 수행 속도가 느리고, 컴파일러 방식은 프로그래밍 작업 당시 번거로움이 있으나 더 빠른 속도로 프로그램을 실행하는 장점이 있다.



<Interpreter 방식>



<Compiler 방식>

	인터프리터 방식	컴파일러 방식
장점	작업 용이하다.	수행 속도가 빠르다.
단점	수행 속도가 느리다.	개발 작업이 번거롭다.
사용 언어	BASIC, JAVA	C, C++, JAVA

6.2. Compiler and Linker

C 는 컴파일러 방식을 사용한다. 여기서는 원시 코드를 196용 컴파일러로 컴파일하는 방법을 살펴보도록 한다.

6.2.1. Compiler

C Compiler 인 ic96의 사용법을 알아보자.

```
[cpath] ic96 [spath] sfile [controls]
```

1>cpath : 컴파일러가 있는 디렉토리 패스
2> spath : 소스 파일이 있는 디렉토리 패스
3> sfile : 소스 파일 이름
4> controls : 컴파일러 컨트롤

ex1> c:> ic96 main.c
→ main.c 소스 파일을 컴파일해서 main.obj 오브젝트 파일을 만든다.
ex2> c:>c:\ic96\ic96 program\main.c md(kc) ot(0) db
→ program 디렉토리에 있는 main.c 소스 파일을 ic96 디렉토리에 있는 컴파일러로 model(kc), optimize(0), debug 컨트롤을 갖도록 컴파일한다.

1) Compiler Control

컨트롤에는 primary, general, invocation –only 세 종류가 있으며, 소스프로그램에서 또는 ex2와 같이 컴파일러를 기동할 경우에 지정할 수 있다.

primary : 컴파일러를 기동할 때 혹은 소스 프로그램에서 preprocessor 지시어 #pragma를 이용하여 처음 실행할 문장 앞에 혹은 소스 문장에서 데이터 정의 문장에 지정할 수 있다. 이 컨트롤은 전체 모듈과 변경하지 않을 부분 혹은 소스 문장에서 일부분을 잠시 중지할 경우 등에 응용할 수 있다.

general : 컴파일러를 기동할 때 혹은 소스 프로그램 전체에서 preprocessor 지시어 #pragma로 사용할 수 있다.

invocation –only : 컴파일러를 기동할 때 사용하며, 전체 모듈에 혹은 컨트롤 인수를 컨트롤하는 데 사용한다.

2) Control의 종류

< "80196" – 차영배 저, 다다미디어 - 참조 >

6.2.2. Linker

C 언어를 C compiler를 이용하여 compile이 끝나면 절대 파일을 만들어야 한다. 이때 linker가 필요하며, 여러 파일을 서로 링크시킬 때에도 linker가 필요하게 된다.

```
[lpath] rl96 [opath] ofile to hfile [controls]
```

1>lpath : 링커가 있는 디렉토리 패스

2> opath : object 파일이 있는 디렉토리 패스

3> ofile : object 파일 이름

4> hfile : 출력 파일 이름

5> controls : 링커 컨트롤

ex1> c:>rl96 main.obj sort.obj debug.obj to sort.out

→ main.obj, sort.obj, debug.obj 의 파일을 서로 링크시켜 sort.out인 object 파일을 만든다.

주의> 1.링크시킬 파일의 확장자를 생략하면 에러가 된다.

2.출력 파일을 지정하지 않으면 처음에 있는 object파일이 출력 파일이 된다.

ex2> c:> rl96 main.obj to main.96

→ main.obj의 출력 파일을 main.96으로 변경한다.

1) Linker Control

1> 리스팅 컨트롤 : 프린트 파일로 보낼 정보가 무엇인지를 지정한다.

2> 링킹 컨트롤 : 출력할 모듈의 이름과 어떤 디버그 정보가 출력 오브젝트 파일에 놓일 거인가를 결정한다.

3> 로케이팅 컨트롤 : Rom, Ram, 레지스터 부분, 리로케터블 세그먼트의 순서를 지정한다.

2) Control 의 종류

< "80196" - 차영배 저, 다다미디어 - 참조 >

6.3. C coding

지금까지 196C의 컴파일 방법과 링크 방법을 살펴보았다. 여기서는 C 프로그램의 기본 구조, C문장, 변수, 상수, 함수 등이 무엇이며 어떤 역할을 하는지, 그리고 조금 더 나아가 제어문에 대해서까지 알아보도록 하자.

6.3.1. C 프로그램의 구성 요소

모든 C 프로그램은 공통된 구성 원소와 성질을 가지고 있다. 즉, 모든 프로그램은 1개 이상의 함수로 구성되며, 각 함수는 1개 이상의 문장을 포함하고 있다. C에서 함수는 이름을 갖고 있는 부프로그램으로 프로그램의 한 부분에서 호출될 수 있다. 하나의 문장은 프로그램에서 수행할 하나의 동작을 나타낸다.

C 함수의 가장 간단한 형태는 다음과 같다.

```
Function-Name ( )
{
  Statement sequence
 }
```

위에서 Function-Name 은 함수 이름이고, statement sequence는 1개 이상의 문장을 의미한다.

6.3.2. C 문장

모든 C 문장은 세미콜론(;)으로 끝난다. C는 한 줄의 끝을 한 문장의 끝으로 간주하지 않는다. 이는 프로그램 작성시, 문장들의 위치에 대한 제약이 없음을 의미하며, 한 줄에 두 개 이상의 문장을 쓸 수 있다. 또한 C는 알파벳의 대문자와 소문자가 구별된다. 따라서 Add 와 add는 서로 다른 이름이다.

6.3.3. 변수 선언과 값의 치환

변수는 다양한 값을 저장할 수 있는 이름이 부여된 기억 장소이다. C에서는 모든 변수들은 사용하기 전에 미리 선언해야 하는데, 변수의 선언은 컴파일러에게 어떤 변수형이 사용되는가를 알려준다. C에서는 5가지 기본 자료형이 제공되며, 다음 표에 키워드와 함께 의미가 설명되어 있다.

자료형	의미	키워드
Character	문자 데이터	char
Integer	정수	int
Float	실수	float

Double	실수	double
Void	값이 없음	void

(float과 double의 차이는 double 이 float 보다 유효숫자의 표현에서 2배의 정밀도를 가지고 있다.)

196 컴파일러의 경우 변수의 타입별 크기와 범위는 다음과 같다.

데이터 타입	크기(바이트)	범 위
char	1	-128~127
unsigned char	1	0~255
int	2	-32,768~32,767
unsigned int	2	0~65,535
long	4	-2,147,483,648~2,147,483,647
unsigned long	4	0~4,294,967,295
float	4	$8.43x10^{-37}$~$3.37x10^{38}$
double		

변수를 선언하기 위하여 다음과 같은 형식을 사용한다.

```
type variable-name;
```

위에서 type는 C자료형이고 variable-name 은 변수의 이름이다.

ex1> int counter;,

→ int형 변수인 counter 선언.

ex2> float x, y, z;

→ float 형 변수인 x, y, z 선언.

변수에 값을 저장하는 것을 치환이라 하며 이를 위하여 등호의 왼쪽에는 변수 이름을, 오른쪽에는 저장하려는 값을 쓴다. 일반적인 치환문의 형식은 다음과 같다.

```
Variable-name =value;
```

ex1> counter = 100 ;

→ counter 라는 정수형 변수에 100을 치환.

ex2> name = 'a' ;

→ name 이라는 문자형 변수에 a 를 치환.

ex3> x = 10.3 ;

→ x 라는 실수형 변수에 10.3을 치환.

주의> 문자 상수는 작은 따옴표 사이에 문자를 쓴 것으로 'A'처럼 나타낸다. 정수형 상수는 그대로 수를 쓰며, 실수 상수는 소수점과 함께 수를 쓴다. 예를 들어 100은 정수 상수이지만. 100.0은 실수 상수이다.

변수를 선언할 때 치환도 함께 할 수 있는데 이를 변수의 초기화라 한다.

ex> int x = 5;

→ 정수형 변수 x를 선언하고 x에 5를 치환함.

변수를 선언하는 곳은 크게 두 군데로 나눌 수 있다. 한 곳은 함수의 내부이고, 다른 한 곳은 모든 함수의 외부이다. 모든 함수의 외부에서 선언된 변수를 전역 변수라 하고 프로그램의 어떤 함수에서도 사용할 수 있고, 프로그램이 실행되는 동안 계속해서 존재한다. 반면, 함수의 내부에 선언된 변수를 지역 변수라 하며 , 지역 변수는 그 변수가 선언된 함수 내에서만 사용할 수 있다. 이런 지역변수는 2가지 특징이 있다. 첫째, 함수 내에 있는 지역 변수들은 다른 함수 내에 있는 지역 변수와 아무런 연관성이 없다는 점이다. 즉 f1란 함수에 있는 counter 라는 변수는 f2란 함수에 있는 counter 와 서로 다는 변수이다. 둘째, 지역 변수는 함수가 호출될 때 생성되고 함수가 종료되면 지역변수는 없어진다. 그러므로 지역 변수들은 함수가 종료된 후 다시 그 함수가 호출될 때 이전의 값을 그대로 보관하지 않는다.

ex>

```
int x ;
void f1()
{
  int y;
  int counter;
}
void f2()
{
  int z;
  int counter;
}
```

→ 여기서 x는 전역 변수, y와 z, counter는 지역 변수로 사용되었다. 또, f1과 f2에 counter 라는 같은 이름을 가진 변수가 존재하는데, 이 두 변수는 서로 연관성이 없다.

6.3.4. 산술식을 이용한 계산

C에서 산술식은 5가지 산술 연산자를 정의한다.

연산자	+	-	*	/	%
의미	덧셈	뺄셈	곱셈	나눗셈	나머지

위에서 +,-,*,/ 연산자는 5가지 기본 자료형과 함께 사용할 수 있다. 그러나 나머지 연산자인 %는 오직 정수형과 함께 사용할 수 있으며 , 정수들의 나눗셈에서 나머지만을 구해준다. 이 연산자가 실수형과 함께 사용되면 아무것도 얻을 수 없다.

식은 치환문의 오른쪽에 둔다. 예를 들어, 다음 문장은 answer라는 정수형 변수에 100*31의 값을 치환한다.

ex> int answer;
answer = 100*31 ;

*,/,% 연산자는 +,- 연산자보다 더 높은 우선 순위를 갖고 있다. 만약 계산 순위를 변경하고 싶다면, 괄호를 사용하면 된다.

ex> 10-2*5 (결과값 0)
(10-2)*5 (결과값 40)

6.3.5. 프로그램에 주석 달기

주석은 프로그램에 기재해 놓은 설명문이다. 모든 주석문은 컴파일러에 의해 무시되며, 오로지 프로그램을 쉽게 이해하기 위한 목적으로 사용된다.

C에서 주석은 /* 과 */ 사이에 기재한다. 예를 들면, 다음과 같이 주석을 설정할 수 있다.

```
/*  this is a comment. */
```

다음과 같이 여러 줄로도 쓸 수 있다.

```
/* this is a longer comment
  Sigma Intelligence */
```

주석은 프로그램의 코드 중 일부를 일시적으로 제거하는데 사용할 수도 있다. 이런 경우 그 부분에 주석 부호를 붙이기만 하면 된다.

6.3.6. 함수

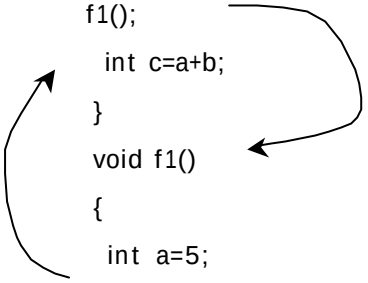
앞에서 C 프로그램은 하나 이상의 함수를 가지고 있다고 했다. 일반적으로 C 프로그램은 많은 함수를 가지고 있고, 이런 함수들이 프로그램 내에서 서로 연관되어 작업을 수행한다고 할 수 있다. C는 기본적으로 하나의 main()함수를 반드시 갖는데, 이 main()함수는 프로그램에서 맨 먼저 실행되는 함수이다. 즉, 프로그램이 시작될 때, main()에 포함된 문장들이 실행되는데, 이 문장들은 '{'과 '}' 내에 포함된다.

1) 함수의 호출

전형적인 함수들은 프로그램 진행 도중 호출되거나 생성된다. 프로그램은 함수를 만날 때까지 한 줄씩 실행하다가 함수를 만나면 그 함수로 분기가 된다. 그 함수가 끝나면 다시 제어권은 그 함수를 불렀던 위치의 다음 줄로 넘겨준다.

ex> main() 실행순서

```
{
 int a=10;                    1
 int b=5;                     2
 f1();
   int c=a+b;                 4
 }
 void f1()
 {
   int a=5;                   3
}
```



2) 함수값의 반환

C에서 함수는 그 함수의 호출 부분으로 값을 반환할 수 있는데, 이를 위해서는 return문을 사용한다. 함수값을 반환할 때는, 반환되는 자료형을 함수 이름 앞에 써준다. 이 때 return 에서 반환하는 자료형과 함수 이름 앞에 선언되는 자료형이 일치해야 한다.

```
return-type function-name()
  {
     return value;
  }

ex> void main()
{
 int num;
 num = f1();
}
int f1()
{
  return 10;
}
```

위에서 f1은 하나의 정수값(10)을 반환한다. 한 가지 중요한 사실로, 함수가 return 문을 만나면 그 함수는 즉각 복귀한다. 즉, return 문 다음에 있는 어떤 문장도 실행되지 않는다.

3) 함수의 인수 사용

한편 함수를 호출할 때 그 함수에게 전달하는 값을 인수라 한다. C에서 함수는 인수를 전혀 갖지않거나, 여러 개의 인수를 가질 수 있다. 함수가 인수를 받기 위해서는 인수 값을 받기 위한 특별한 변수를 선언해야만 한다. 이것을 함수의 형식 매개변수 혹은 매개변수라 부른다. 매개변수는 함수의 이름 바로 다음에 괄호 안에서 선언된다.

```
function-name(parameter-type variable-name,・・・ )
{
 Statement sequence;
}
```

다음은 2개의 정수 인수들의 합을 반환하는 함수이다.

```
ex> main()
{
 int a, b;
 a = sum (1,2);
b = sum (5,6);
}
int sum(int x, int y)
{
 return x+y;
}
```

sum()이란 함수를 호출할 때마다, 이 함수는 x에 전달된 값과 y에 전달된 값을 더한다. 이 프로그램은 a, b에 3과 11이란 값을 반환한다. sum()이 호출될 때, C 컴파일러는 각 인수의 값을 대응 하는 매개변수에 복사한다. 즉 sum()의 첫번째 호출에서는 1이 x에 복사되고, 2는 y에 복사된다. 두번째 호출에서는 5가 x에 복사되고, 6이 y에 복사된다.

6.3.7. 제어문

프로그램에서 제어문은 프로그램이 수행되는 흐름을 제어하므로, 프로그램의 주축이 된다. 여기서는 C언어의 가장 중요한 제어문이라 할 수 있는 if 와 for 문을 살펴볼 것이다.

1) if 조건문

if문은 C언어의 조건문 중 하나이다. if 문의 수행은 조건이 참인지 거짓인지에 따라 실행문이 결정된다. 간단히 말해서 선택문들은 조건의 결과에 따라 수행여부를 결정한다. 이러한 if 문의 형식은 다음과 같다.

```
if (expression) statement;
```

위의 형식에서 식(expression)의 결과가 참이면 문장(statement)가 실행되고, 거짓이라면 문장은 실행되지 않고, if 문의 다음 문장들을 실행하게 된다. C에서는 0이 아닌 값은 무조건 참이 되고, 0이면 거짓이 된다.

일반적으로 if 문의 내부의 식은 하나의 값과 다른 값을 비교하기 위해 관계 연산자를 사용한다. 가장 많이 쓰이는 관계 연산자를 먼저 소개하면, <,>,==이다. <과 >는 양변의 크기를 비교할 때 쓰이고, ==는 값이 같음을 검사하기 위해서 쓰인다.

ex1> if (5>4) a=7;

→ 이 경우 5>4인 식의 값이 참이므로 a에 7을 치환한다.

ex2> if(5<4) a=7;

→ 이 경우 5<4 인 식의 값이 거짓이므로 a=7이란 문장을 수행하지 않고 넘어간다.

if에 else 문을 추가할 수 있다. else가 추가되면 if 문의 형식은 다음과 같이 된다.

```
if (expression) statement1;
    else statement2;
```

if 문에 있는 식(expression)이 참이면, if 문의 statement1이 수행되고, else문에 있는 statement2는 실행되지 않는다. 그러나, 식이 거짓이면 if문의 statement1은 실행되지 않고, else문의 statement2가 실행된다. 어떠한 경우라도 두 문장 모두 실행될 수 없고, 두 가지 경로 중 한 경로만 선택된다.

if 문에서 수행해야 할 문장이 많을 때에는 블록(block)을 사용한다. 블록은 문장들의 시작과 끝에 중괄호를 사용하여 만든다. 블록을 만들고 나면, 문장들은 단일 문장이 사용되는 것처럼 된다.

```
ex> if(num>0) {
        a=x+y;
        b=x*y;
    }
```

2) for 반복문의 사용

어떤 문장을 반복하고자 할 때 가장 많이 쓰이는 반복문이 for 반복문이다. 가장 일반적인 형식의 for 문은 지정된 수만큼 문장이나 블록을 반복하는데 사용된다. 단일 문장을 반복하는 일반적인 형식은 다음과 같다.

```
for (initialization; conditional test; increment) statement;
```

초기화(initialization) 부분은 반복을 제어하는 변수에 초기값을 설정하는 부분인데, 이 변수를 반복 제어 변수라 부른다. 초기화 부분은 반복이 시작되기 전에 한 번만 실행된다. 조건 검사(conditional test)부분은 반목할 때마다 목표값에 대하여 반복-제어 변수를 검사한다. 조건 검사의 결과가 참이면 반복은 계속되고, 조건 검사의 결과가 거짓이면 반복은 거기서 멈추고, 반복문 다음에 오는 문장들을 실행하게 된다. 조건 검사는 반복할 때마다 반복의 첫 부분에서 실행된다. for문의 증가 연산은 매 반복의 끝에서 실행된다. 증가 연산은 문장이나 블록이 실행되고 나서 실행되나, 조건 검사의 전에 실행된다.

다음은 1부터 10까지 더하는 프로그램이다.

```
ex> main()
{
 int sum = 0;
 for (int num = 1; num<11; num=num+1) sum=sum+num;
 }
```

여러 개의 문장을 반복하려면 for 반복의 목표문을 블록으로 묶어주면 된다.

```
ex> main()
{
 int sum = 0;
  int prod=1;
 for (int num = 1; num<11; num=num+1)
   { sum=sum+num;
     prod=prod * num;
     }
 }
```

일반적으로 for 문에 대해서, 반복의 증가 연산은 num=num+1 과 같은 형태를 많이 하고 있다. C에는 변수의 값을 1만큼 증가시키는 연산자가 제공되므로, 대부분의 프로그램에서는 num=num+1이라는 문장을 거의 쓰지 않는다. 증가 연산자는 ++ 인데, 위의 num=num+1 을 증가 연산자를 사용하여 바꾸면 num++ 로 쓸 수 있다. 마찬가지로 감소 연산자는 –-로 num--

와 같이 쓴다.

3) 관계와 논리 연산자

앞에서 관계 연산자에 대해 간단히 살펴보았다. 여기서는 앞에서 다룬 관계 연산자를 비롯하여, 논리 연산자에 대해서 알아보자. 관계 연산자와 논리 연산자의 종류는 다음과 같다.

관계 연산자				논리 연산자	
연산자	의미	연산자	의미	연산자	의미
>	크다	<	작다	&&	AND
>=	크거나 같다	<=	작거나 같다	||	OR
==	같다	!=	같지 않다	!	NOT

연산자의 우선 순위는 다음과 같다.

가장 높음 !

> >= < <=

== !=

&&

가장 낮음 ||

6.4. Examples

지금까지 C프로그래밍에 대해 살펴보았다. 이번에는 몇 가지 예제를 보면서 지금까지 배운 C 문법을 실제로 어떻게 사용하는지 알아보자.

6.4.1. LED 점멸 프로그램

```
#include "80c196kc.h"  /* 80c196kc.h 라는 파일을 읽음 */
/* led 점멸을 위한 지연함수 */
void delay(int i);
{
        for(;i--;);
}
```

```
 /* 메인 함수  */
void main()
{
        unsigned char led;
        led = 0xfe;                 /* led 점등 초기값 */
        for (;1;)           /* 무한 루프 */
        {
                ioport1=led;       /* 포트1로 출력 */
                led = (led<<1) | 1;/* 왼쪽으로 한비트 이동 */
                if ((led & 0x10)==0) led = 0xfe ;       /* led 점멸이 끝인지 체크 */
                delay(3000);       /* led on 시간 */
        }
 }
```

6.4.2. 스텝모터 구동 프로그램

```
 #include "80c196kc.h"
 data unsigned char pointer, a;
const data unsigned char r_pls_table[] = {0x90, 0x80, 0xa0, 0x20, 0x60, 0x40, 0x50, 0x10};
const data unsigned char l_pls_table[] = {0x09, 0x08, 0x0a, 0x02, 0x06, 0x04, 0x05, 0x01};
void delay(int time);
 void main(void)
{
        pointer = 0;
         for(;1;)   /* 무한 루프  */
         {
                  pointer++;
                pointer &= 0x07;
                a = r_pls_table[pointer] | l_pls_table[pointer];
                ioport1 = a;
                 delay(5);
        }
}
 void delay(int time)    /* 모터 구동을 위한 지연 함수 */
{
        int a,b;
         for(a=0;a<time;a++)
```

```
                for(b=0;b<time;b++){}
}
```

6.4.3. 80c196kc.h

```
 #pragma model(kc)
#pragma code
 #include "kc_funcs.h"
#include "kc_sfrs.h"
 #pragma interrupt(nmi=31)
#pragma interrupt(hsi_fifo_full=30)
#pragma interrupt(extint1=29)
#pragma interrupt(timer2_overflow=28)
#pragma interrupt(timer2_capture=27)
#pragma interrupt(hsi_fifo_4=26)
#pragma interrupt(receive_interrupt=25)
#pragma interrupt(transmit_interrupt=24)
#pragma interrupt(unimplemented_opcode=9)
#pragma interrupt(trap_int=8)
#pragma interrupt(extint=7)
#pragma interrupt(serial_timer=6)
#pragma interrupt(software_timer=5)
#pragma interrupt(hsi_0_pin=4)
#pragma interrupt(high_speed_outputs=3)
#pragma interrupt(hsi_data_available=2)
#pragma interrupt(ad_conversion_complete=1)
#pragma interrupt(timer_overflow=0)
```

6.4.4. kc_sfrs.h

```
 /*
 * kc_sfrs.h
 * Generated with gen-sfrs.pl 1.4
 */
 #ifndef _kc_sfrs_h
#define _kc_sfrs_h
 /*                                                    */
```

```
/* Special function register definitions for the                    */
/*                                                                  */
/* 8XC196KC                                                         */
/*                                                                  */
/* Generated from @(#)kc_sfrs.db  1.7                               */
/*                                                                  */
/* Copyright (C) 1994 Tasking Software B.V.                         */
/*                                                                  */
 /* Hwindow 0                                                       */
extern volatile unsigned short register r0;                  /* 0x0000: R/W */
extern volatile unsigned short register zero_reg;            /* 0x0000: R/W */
extern volatile unsigned short register ad_result;           /* 0x0002: R   */
extern volatile unsigned char  register ad_result_lo;        /* 0x0002: R   */
extern volatile unsigned char  register ad_result_hi;        /* 0x0003: R   */
extern volatile unsigned char  register ad_command;          /* 0x0002: W   */
extern volatile unsigned char  register hsi_mode;            /* 0x0003: W   */
extern volatile unsigned short register hsi_time;            /* 0x0004: R   */
extern volatile unsigned short register hso_time;            /* 0x0004: W   */
extern volatile unsigned char  register hsi_status;          /* 0x0006: R   */
extern volatile unsigned char  register hso_command;         /* 0x0006: W   */
extern volatile unsigned char  register sbuf;                /* 0x0007: R/W */
extern volatile unsigned char  register sbuf_rx;             /* 0x0007: R   */
extern volatile unsigned char  register sbuf_tx;             /* 0x0007: W   */
extern volatile unsigned char  register int_mask;            /* 0x0008: R/W */
extern volatile unsigned char  register int_pend;            /* 0x0009: R/W */
extern volatile unsigned char  register int_pending;         /* 0x0009: R/W */
extern volatile unsigned short register timer1;              /* 0x000a: R   */
extern volatile unsigned char  register watchdog;            /* 0x000a: W   */
extern volatile unsigned char  register ioc2;                /* 0x000b: W   */
extern volatile unsigned short register timer2;              /* 0x000c: R/W */
extern volatile unsigned char  register ioport0;             /* 0x000e: R   */
extern volatile unsigned char  register baud_rate;           /* 0x000e: W   */
extern volatile unsigned char  register ioport1;             /* 0x000f: R/W */
extern volatile unsigned char  register ioport2;             /* 0x0010: R/W */
extern volatile unsigned char  register sp_stat;             /* 0x0011: R   */
extern volatile unsigned char  register sp_con;              /* 0x0011: W   */
```

```
extern volatile unsigned char   register ipend1;              /* 0x0012: R/W */
extern volatile unsigned char   register int_pend1;           /* 0x0012: R/W */
extern volatile unsigned char   register imask1;              /* 0x0013: R/W */
extern volatile unsigned char   register int_mask1;           /* 0x0013: R/W */
extern volatile unsigned char   register wsr;                 /* 0x0014: R/W */
extern volatile unsigned char   register ios0;                /* 0x0015: R   */
extern volatile unsigned char   register ioc0;                /* 0x0015: W   */
extern volatile unsigned char   register ios1;                /* 0x0016: R   */
extern volatile unsigned char   register ioc1;                /* 0x0016: W   */
extern volatile unsigned char   register ios2;                /* 0x0017: R   */
extern volatile unsigned char   register pwm0_control;       /* 0x0017: W   */
extern volatile unsigned short register sp;                   /* 0x0018: R/W */
/* Hwindow 1                                                              */
extern volatile unsigned char   register ad_time;             /* 0x0003: R/W */
extern volatile unsigned short register ptssel;               /* 0x0004: R/W */
extern volatile unsigned short register ptssrv;               /* 0x0006: R/W */
extern volatile unsigned char   register ioc3;                /* 0x000c: R/W */
extern volatile unsigned char   register pwm1_control;        /* 0x0016: R/W */
extern volatile unsigned char   register pwm2_control;        /* 0x0017: R/W */
/* Hwindow 15                                                             */
extern volatile unsigned short register t2capture;            /* 0x000c: R/W */
#endif /* _kc_sfrs_h */
```

7. Applications

7.1 Line Tracer

라인 트레이서는 바닥에 있는 줄을 따라 움직이는 로봇이다. 프로그램을 어떻게 구현하느냐에 따라 다르겠지만, 흰 바탕에 검은 줄을 따라 갈 수도 있고, 검은 바탕에 흰 줄을 따라 가게 할 수 있다. 이번 신입생 교육에서는 흰 바탕에 검은 줄을 따라 가는 라인 트레이서를 만들 계획이다.

7.2 Obstacle Racer

Obstacle Racer는 앞으로 가다가 앞에 바닥이 없으면 옆으로 꺾어서 가는, 지하철 입구 등에서 흔히 볼 수 있는 로봇이다. 라인 트레이서와 같이 센서를 사용하는 로봇으로, 앞에 바닥이 없을 경우 왼쪽이나 오른쪽으로 방향을 바꾸는 방법으로 알고리즘을 구현할 수 있다.

7.3 Control Car

Control Car는 Line Tracer나 Obstacle Racer처럼 센서를 사용하여 방향을 조정하는 방법이 아니라, 로봇과 키패드로 이루어진 조종기판이 유선으로 연결 되어있어, 키패드를 통해 사람이 직접 조종할 수 있는 로봇이다.

7.4 etc.

1> 예전(작년까지)에는 cpu를 196이 아니라 z80을 사용했었다. 그 때 나온 작품들을 살펴보면 다음과 같다.

- ♦ 1998년 : Z80 CPU 보드와 8*8 LED Matrix를 이용한 테트리스 게임
  Z80 CPU 보드와 LCD Display장치를 이용한 디지털 시계
- ♦ 1999년 : puzzle 게임, 피아노 음을 저장하고 그음에 맞는 음을 재생하는 건반,
  사람의 음을 녹음하는 voice recoder 4,
  servo motor를 이용하여 4발로 걸어다닐 수 있는 4족 로봇.
- ♦ 2000년 : R/C car , 두더지 게임, LCD를 이용한 쌍뱀 게임

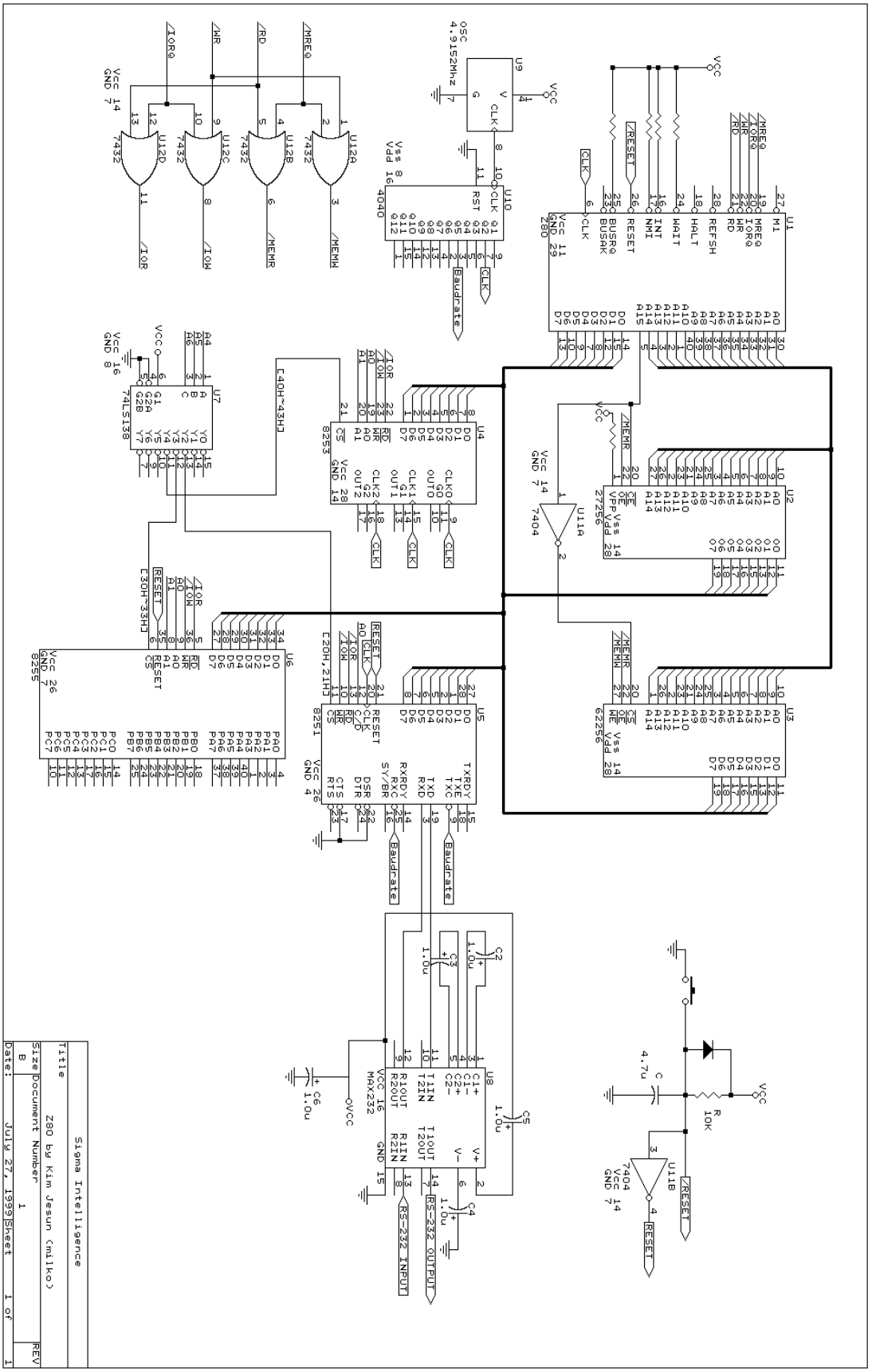
2> 현재 Sigma Intelligence에 있는 양대 프로젝트(로봇축구와 마이크로 마우스)에 참가하여 활동할 수도 있다. 로봇 축구는 예전에 카이스트라는 드라마로 많이 알려져 있는데, 이를

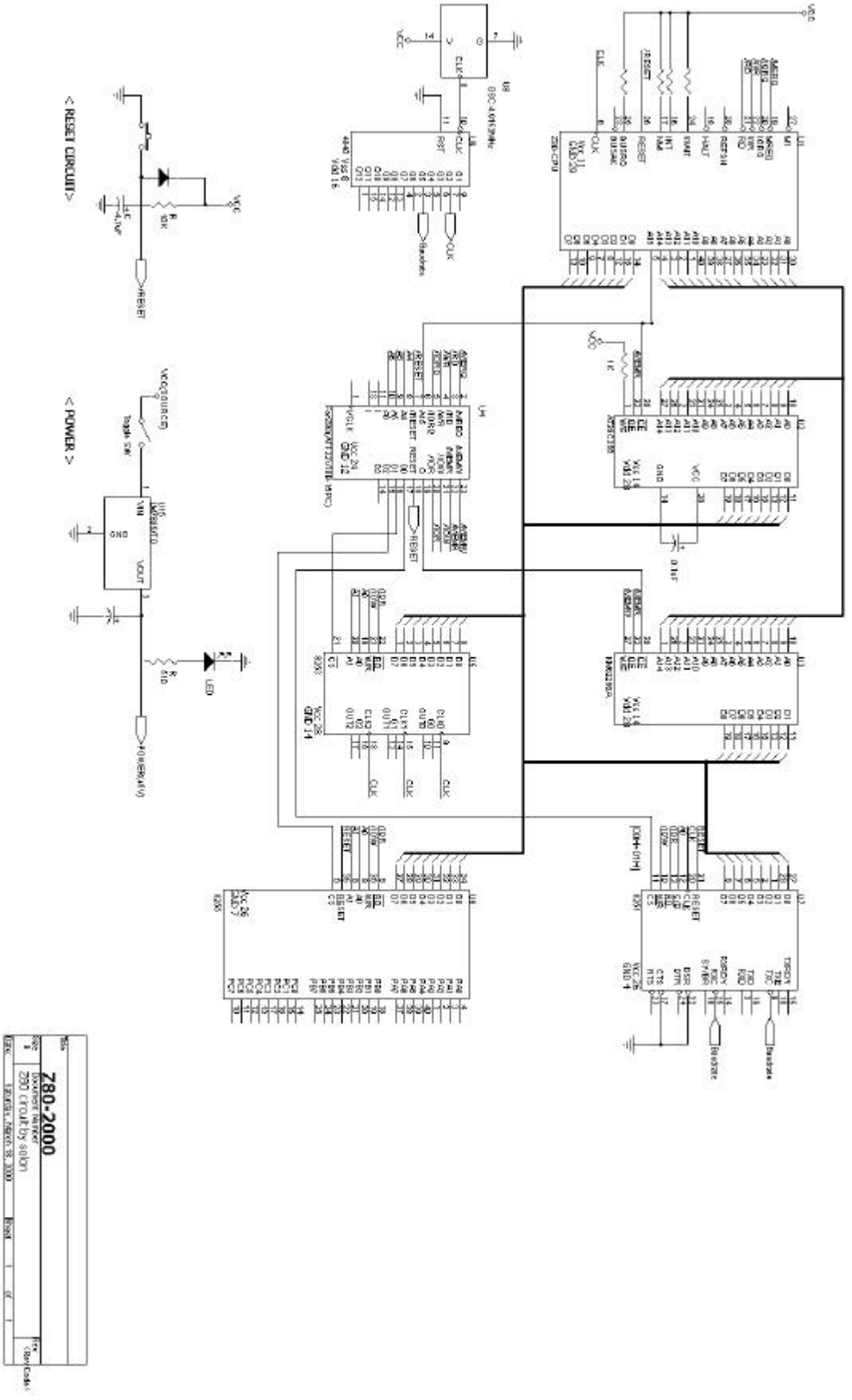
대략적으로 설명하면, 우선 경기장에 있는 로봇을 카메라로 잡아 호스트 컴퓨터에서 로봇과 공의 위치를 파악한다. 호스트 컴퓨터에서 로봇과 공의 위치에 알맞은 작전을 구상하여, 각각의 로봇에 명령을 내리면, 로봇이 그 명령을 수행하는 방식으로 로봇 축구는 이루어 진다.

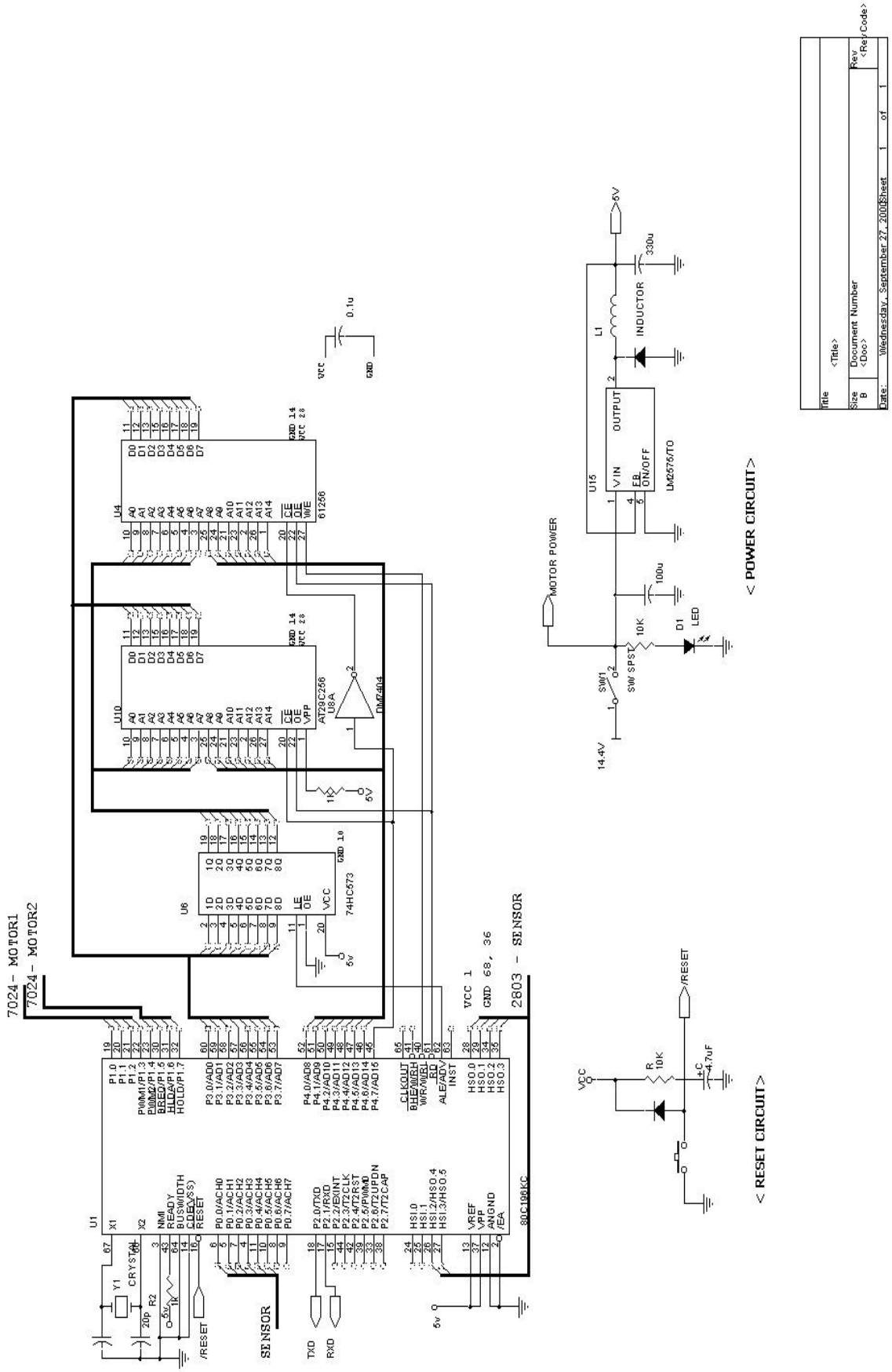
마이크로 마우스는 미로 찾는 로봇이라고 생각하면 가장 쉽게 와 닿을 것이다. 로봇에게 미로의 크기를 알려주면, 로봇은 처음 두 번의 주행에서 미로의 중앙부를 찾고, 중앙부로 가는 몇 가지 길을 기억한다. 다음 세 번째 주행에서는 미로의 중앙부를 가장 빠른 시간에 갈 수 있는 길을 찾아가는 로봇이다.

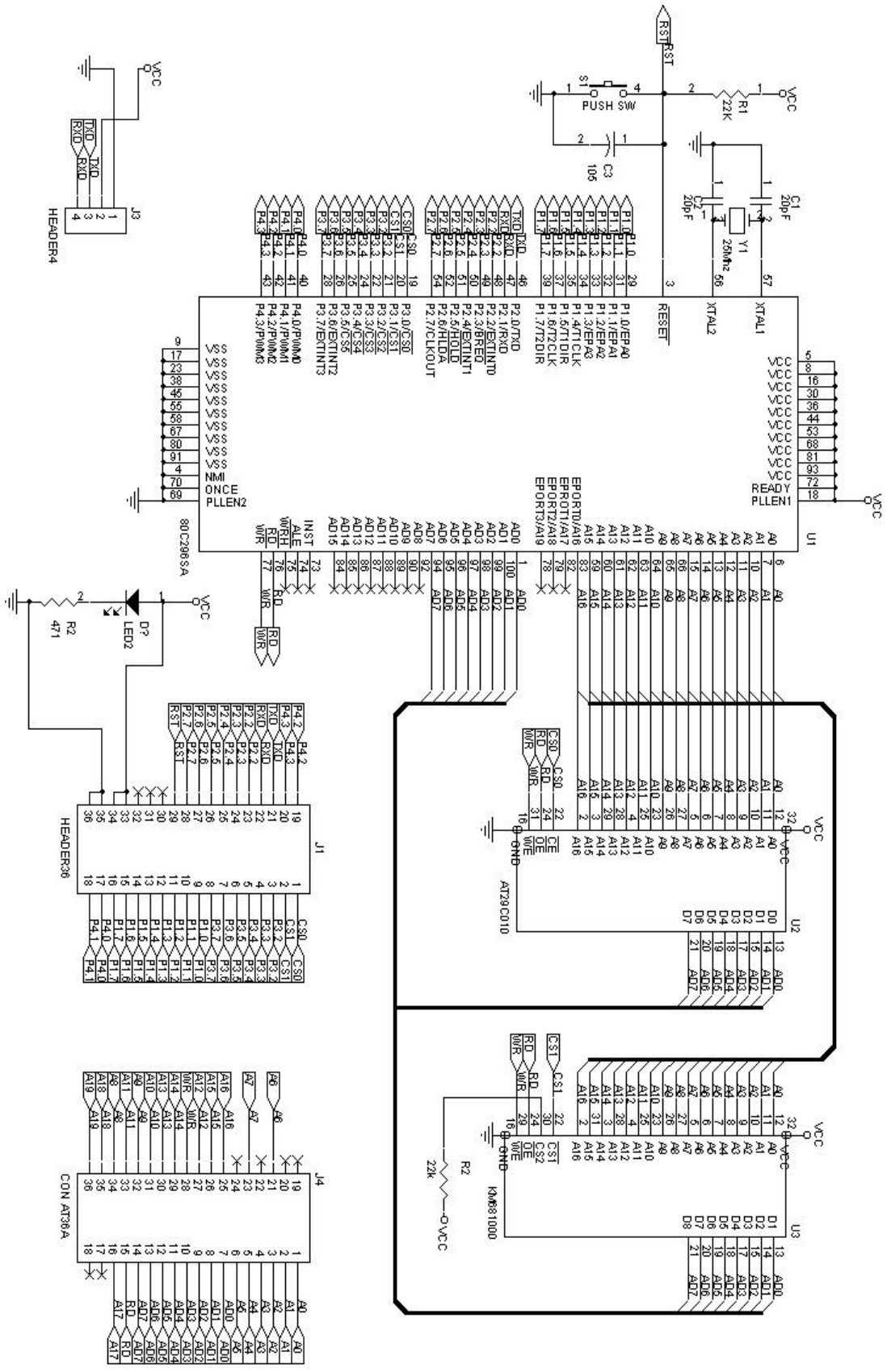
Appendix

A. Schematics
B. Sockets


Sigma Intelligence
Z80 by Kim Jesun (milko)
Baudrate
RS-232 OUTPUT
RS-232 INPUT
/RESET
/MEMR
/MEMW
/IOR
/IOW
/MREQ
/RD
/WR
/IORQ


< RESET CIRCUIT >
< POWER >
Z80-2000
200 circuit by sabin

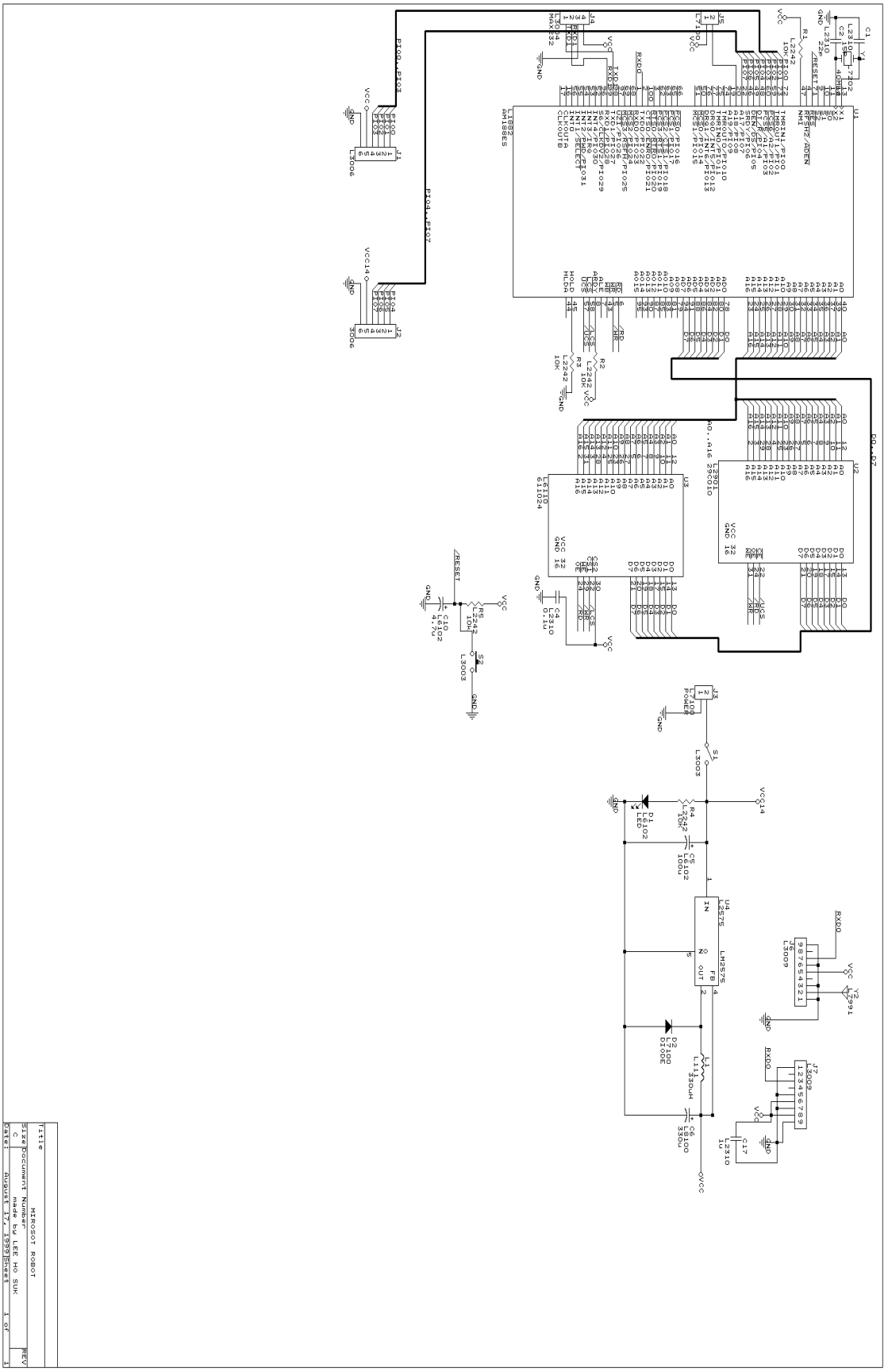

7024 - MOTOR1
7024 - MOTOR2
2803 - SENSOR
SENSOR
TXD
RXD
/RESET
MOTOR POWER
INDUCTOR
LM2576/TO
LED
SW SPST
< POWER CIRCUIT >
< RESET CIRCUIT >
80C196KC
74HC573
AT29C256
61256
Title
<Title>
Document Number
<Doc>
Rev
<Ref Code>
Wednesday, September 27, 2000
Sheet 1 of 1

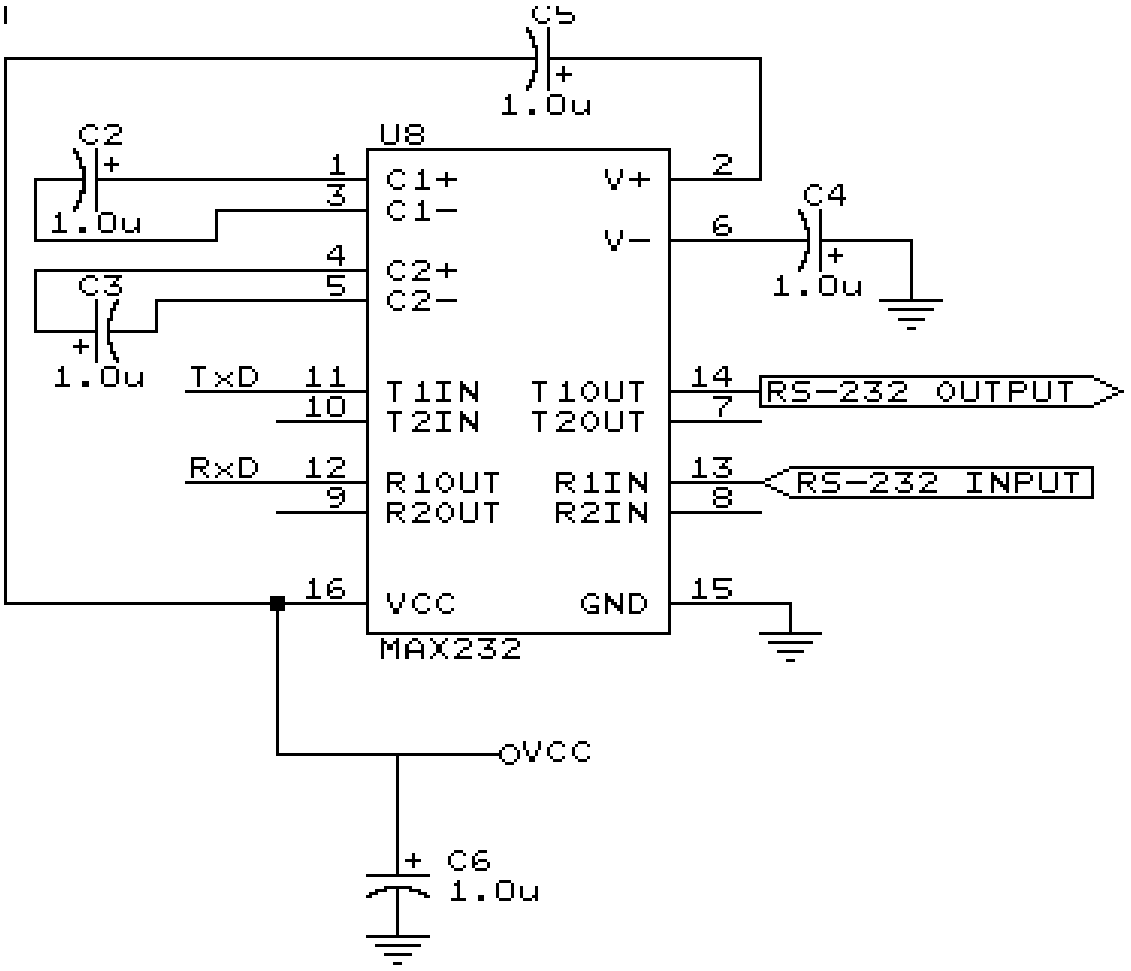

80C296SA
U1
AT29C010
U2
K9F81000
U3
HEADER36
J1
CON AT36A
J4
HEADER4
J3
PUSH SW
S1
22K
R1
105
C3
20pF
C1
20pF
C2
25MHz
Y1
RESET
XTAL1
XTAL2
READY
PLLEN1
NMI
ONCE
PLLEN2
LED2
D?
471
R2
22k
R2
VCC

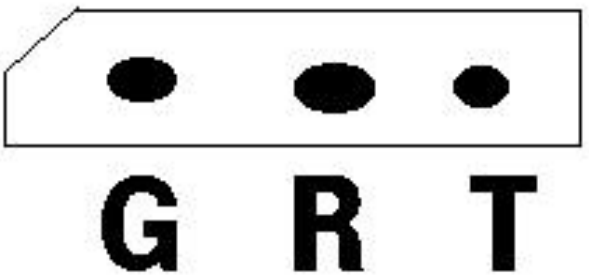
P3.0	P3.1		A0	A1
P3.2	P3.3		A2	A3
P3.4	P3.5		A4	A5
P3.6	P3.7		D0	D1
P1.0	P1.1		D2	D3
P1.2	P1.3		D4	D5
P1.4	P1.5		D6	D7
P1.6	P1.7		RD	A17
P4.0	P4.1		X	X
P4.2	P4.3	80c296 PCB 보드 윗면도	X	X
P2.0	P2.1		A6	X
P2.2	P2.3		A7	X
P2.4	P2.5		A16	A15
P2.6	P2.7		A12	WR
리셋	X		A14	A13
X	X		A10	A9
+	+		A11	A8
−	−		A18	A19

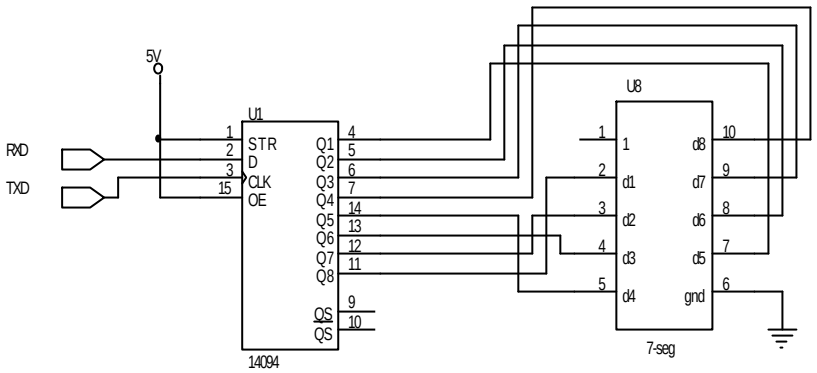
A1	A0		P3.1	P3.0
A3	A2		P3.3	P3.2
A5	A4		P3.5	P3.4
D1	D0		P3.7	P3.6
D3	D2		P1.1	P1.0
D5	D4		P1.3	P1.2
D7	D6		P1.5	P1.4
A17	RD		P1.7	P1.6
X	X		P4.1	P4.0
X	X	80c296 PCB 보드 뒷면도	P4.3	P4.2
X	A6		P2.1	P2.0
X	A7		P2.3	P2.2
A15	A16		P2.5	P2.4
WR	A12		P2.7	P2.6
A13	A14		X	리셋
A9	A10		X	X
A8	A11		+	+
A19	A18		−	−

< 80296SA PCB Header >


Title
MIROSOT ROBOT
Size: C
Document Number:
made by LEE HO SUK
Date: August 17, 1999
Sheet 1 of 1
REV

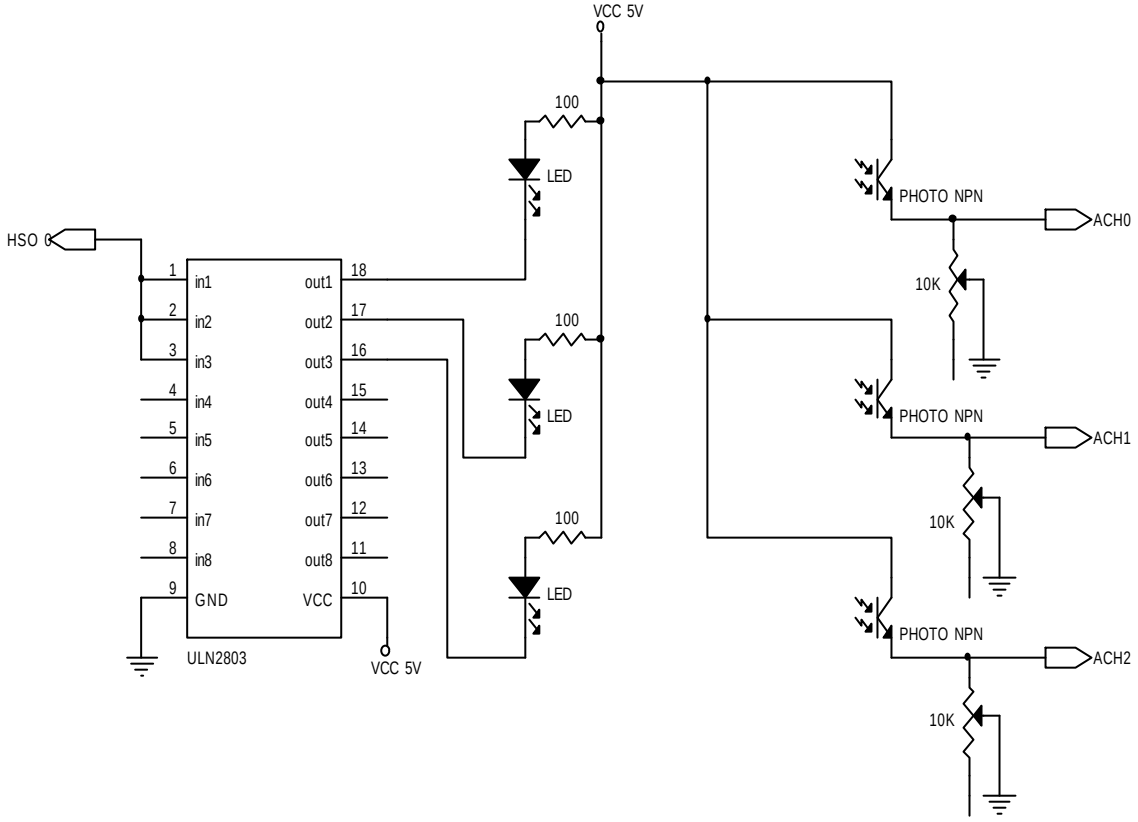

C5
1.0u
C2
1.0u
U8
C1+
C1-
C2+
C2-
V+
V-
C4
1.0u
C3
1.0u
TxD
T1IN
T2IN
T1OUT
T2OUT
RS-232 OUTPUT
RxD
R1OUT
R2OUT
R1IN
R2IN
RS-232 INPUT
VCC
GND
MAX232
VCC
C6
1.0u


G R T

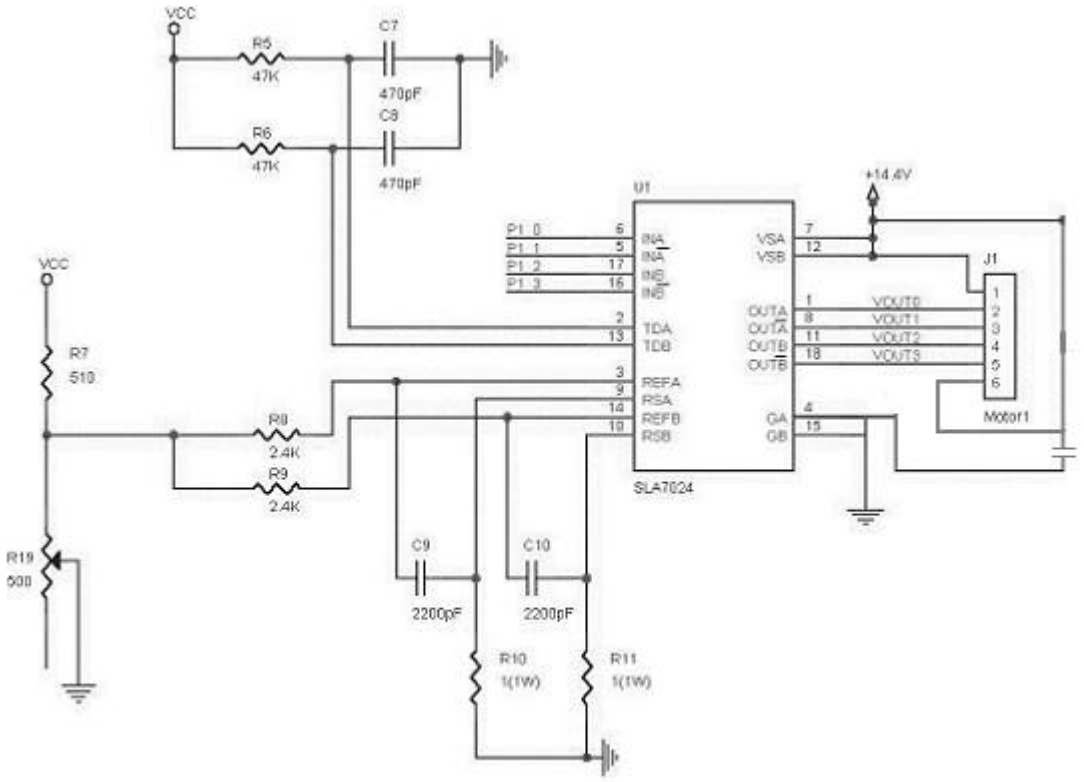


Title <Title>		
Size A	Document Number <Doc>	Rev <RevCode>
Date: Wednesday, September 27, 2000	Sheet 1 of 1	

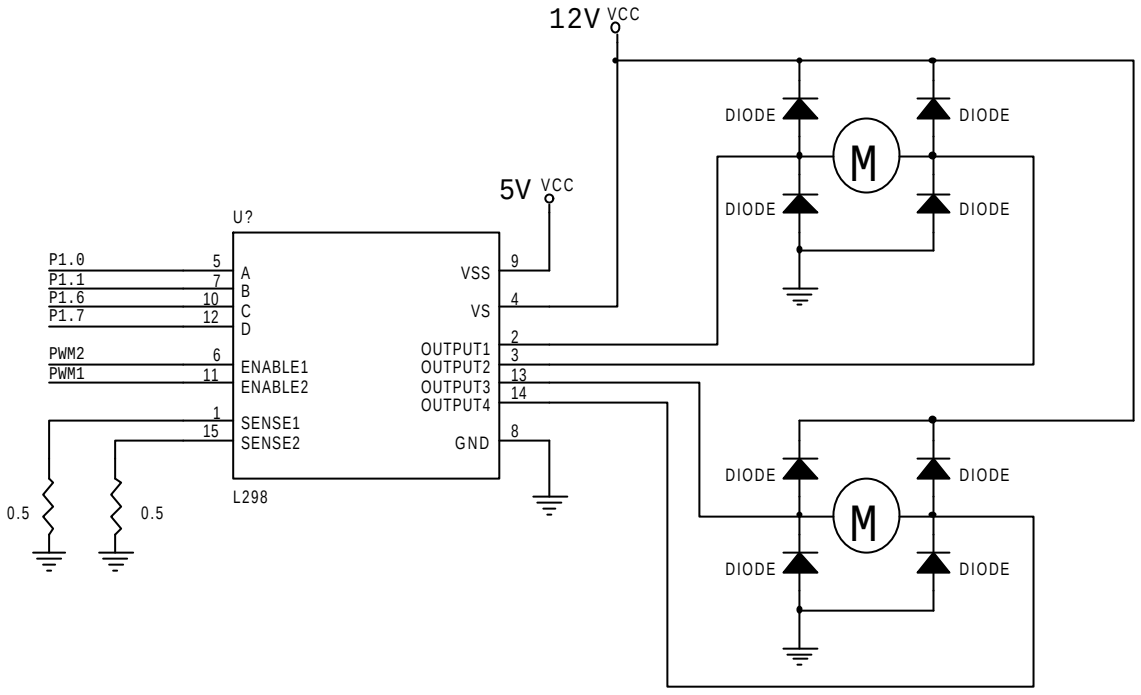
< 7 Segment Driver >



< Micro Mouse Sensor Part >



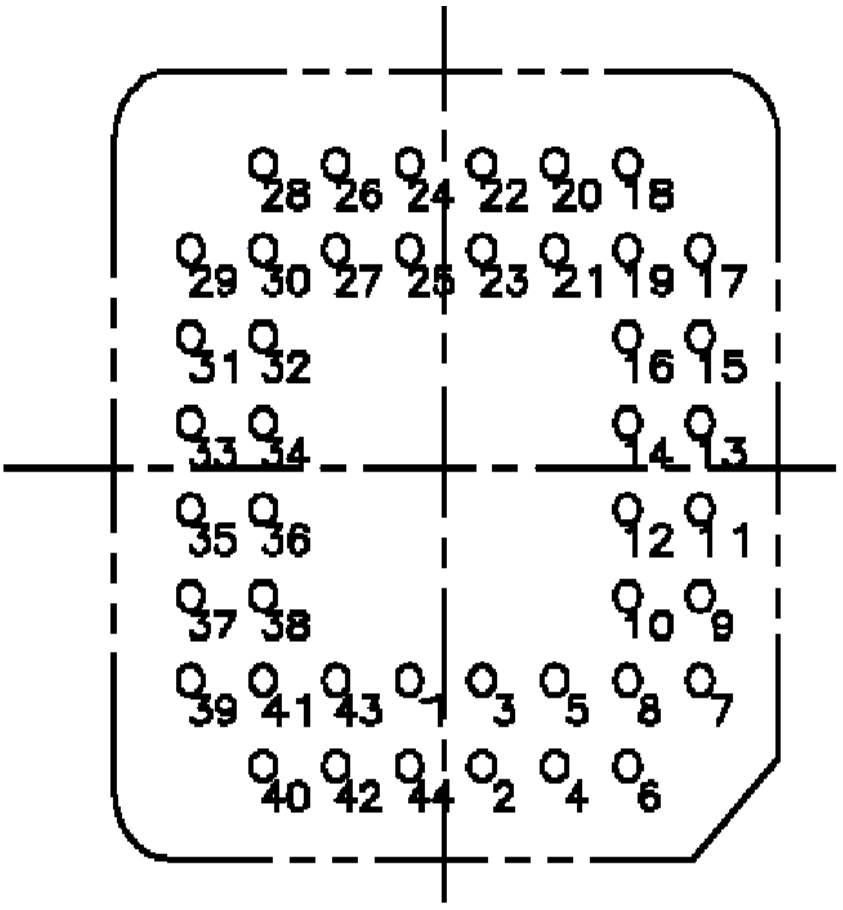
< Stepping Motor Driver >



	left motor(PWM1)			right motor (PWM2)	
	P1.7	P1.6		P1.1	P1.0
turn left	0	1	turn right	0	1
turn right	1	0	turn left	1	0

< DC Motor Driver >

44 Pin



44 POSITION

61 63 65 67 01 03 05 07 09

60 62 64 66 68 02 04 06 08 11 10

58 59 13 12

56 57 15 14

54 55 17 16

52 53 19 18

50 51 21 20

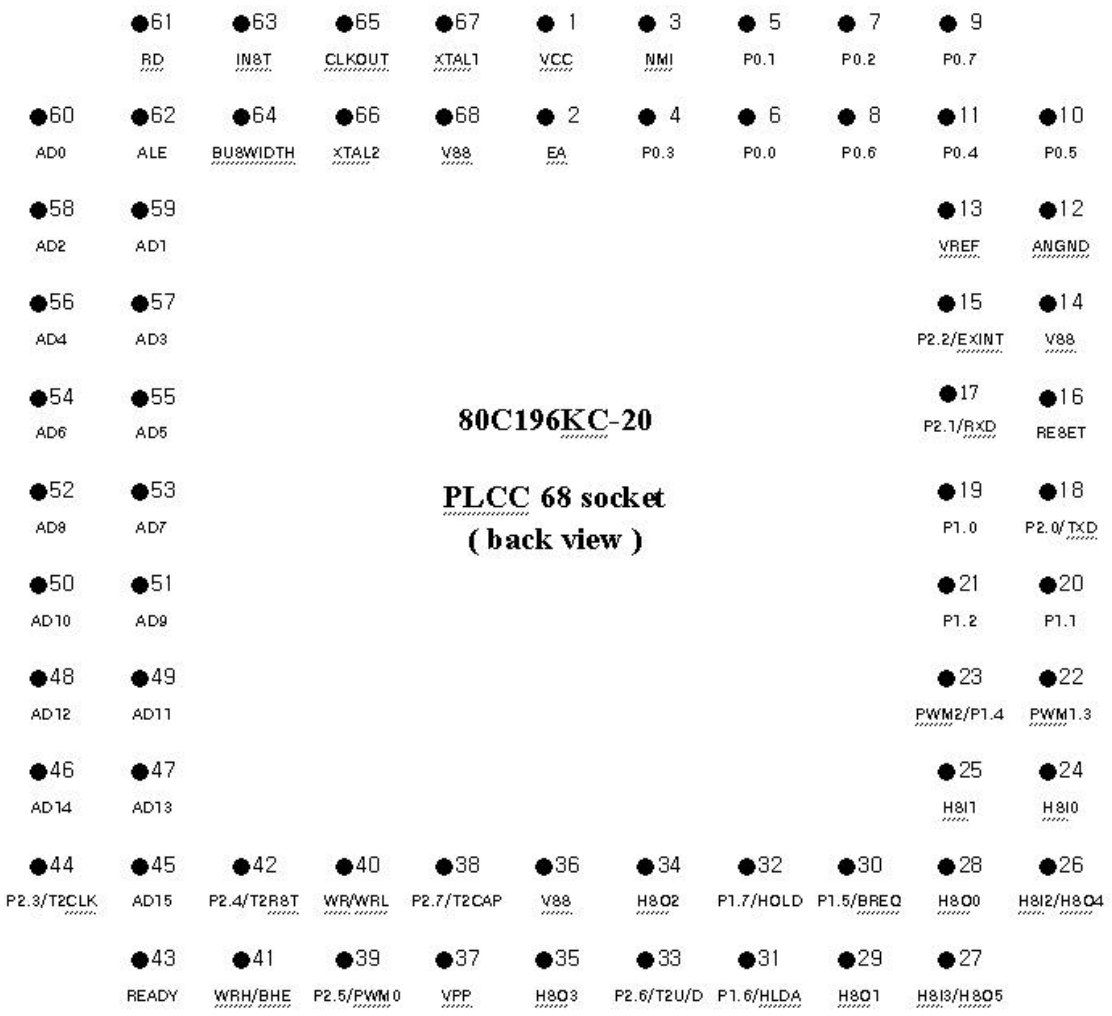
48 49 23 22

46 47 25 24

44 45 42 40 38 36 34 32 30 28 26

43 41 39 37 35 33 31 29 27

< 68 pin PLCC >



< 80C196KC - 68 pin PLCC >

75 77 79 81 83 01 03 05 07 09 11

74 76 78 80 82 84 02 04 06 08 10 13 12

72 73 15 14

70 71 17 16

68 69 19 18

66 67 21 20

64 65 23 22

62 63 25 24

60 61 27 26

58 59 29 28

56 57 31 30

54 55 52 50 48 46 44 42 40 38 36 34 32

53 51 49 47 45 43 41 39 37 35 33

< 84 pin PLCC >

81 83 85 87 89 91 93 95 97 99
82 84 86 88 90 92 94 96 98 100

79 80	2 1
77 78	4 3
75 76	6 5
73 74	8 7
71 72	10 9
69 70	12 11
67 68	14 13
65 66	16 15
63 64	18 17
61 62	20 19
59 60	22 21
57 58	24 23
55 56	26 25
53 54	28 27
51 52	30 29

50 48 46 44 42 40 38 36 34 32
49 47 45 43 41 39 37 35 33 31

< 100 pin PLCC >

Authors

'00 Kim Han-jun

'00 Lee Dong-yun

'00 Lee Moon-hwan

'99 Park Jong-min

'99 Choi Jong-hyun

'99 Hyun Gyoung-hwan

Last revised at 2001. 3. 17.

Edited by Park Jong-min.

Directed by Hyun Gyoung-hwan.