論文95-32B-3-4

Instruction FIFO Memory를 이용한 범용 DSP 구조

(A General Purpose DSP Architecture Using Instruction FIFO Memory)

朴 柱 炫 *, 金 榮 民 *

(Ju Hyun Park, and Young Min Kim)

요 약

본 논문은 FIFO 메모리를 이용한 프로그래머블 16비트 DSP 구조를 연구하여 독창적인 DSP 구조를 제안한다. 또한 본 DSP 구조와 관련하여 system 구성, Bus 구성, Instruction set을 개발하였으며 system test를 위해 어셈블러를 별도로 개발하였다. Hardware의 특징은 명령어를 RAM에서 fetch하지 않고 FIFO로부터 shift 명령만으로 간단히 가져올 수 있기 때문에 RAM 접속 시간에 상관없이 시스템 설계가 가능하다. 또한 명령어 파이프라인이 이루어질 경우 한 명령어를 수행하는데 있어 1 사이클만으로도 충분하게 된다. 또한 파이프라인 수를 줄임으로써 복잡한 제어 장치를 구성하지 않아도 된다는 장점을 가지고 있다.

Abstract

In this paper, we propose a programmable 16 bit DSP architecture using FIFO instruction memory. With this DSP architecture, System structure, BUS structure, instruction set and an assembler for system test are developed. The characteristic of this structure is that it simply fetches instructions not from RAM but from FIFO using shift operations. Accordingly, System can be designed regardless of RAM access time. One cycle is enough to execute an instruction, if instruction pipeline is operated. Another merit of this structure is that we can obtain the same effect as instruction pipelining without constructing a complex pipelined controller by decreasing the pipeline number.

I. 서 론

최근 과학이 발달하고 생활 수준이 높아짐에 따라 사람들은 통신을 생각함에 있어 과거 단순한 정보 전달 수단의 차원을 넘어 더욱 양질의 정보를 원하게 되었다. 특히 digital 신호의 여러 가지 장점들 즉, 잡음에 강한 특성, 저장 및 수정의 용이성, data 조작의 편리함 등 때문에 analog 방식에서 벗어나 digital 방식이 점차 보편화되고 있는데 이에 따라 digital 신호 처리의 중요성 또한 점차 높아져 가고 있다. 현재 많은 연구가 이루어지고 있는 휴대용 전화기, 화상 전송, 영상 회의등이 미래의 정보 산업으로 급속히 부상하고 있으며, 이를 위한 정보 기기들이 양질의 음성이나 화상등을 전송하기 위해서는 더욱 더 많은 양의 digital 신호 처리가 필요하게 된다. 그러므로 이처럼 많은 데이타를 처리하기 위해서는 보다 빠르게 신뢰도 높은 결과치를 얻을 수 있는 DSP(Digital Signal Processing) chip의 구현이 필수적이라 하겠다.[1] 그러

* 正會員, 全南大學校 電子工學科
(Dept. of Elec. Eng., Junnam Nat'l Univ.)
接受日字 : 1994年 9月 13日

나 기존 DSP chip은 한 개의 명령어를 수행하는데 있어 여러 사이클의 과정이 필요할 뿐만 아니라 각 사이클 수행에 걸리는 시간 또한 길다. 따라서 속도를 높이기 위해 RISC(Reduced Instruction Set Computer) 개념을 도입하여 단일 클럭에 단일 명령어를 수행할 수 있도록 하고 있다. 그러나 RISC 구조는 명령어 파이프라인을 해야 하므로 제어 구조가 복잡해질 뿐만 아니라 구조의 단순화를 위해 Load/Store 명령어 위주로 사용하기 때문에 내부 cache를 사용하지 않을 경우 RAM에 접속하는 명령어 비율이 높아진다.

본 연구에서는 기존 RISC의 이러한 단점을 보완하기 위하여 명령어 및 데이타 fetch에 FIFO를 이용함으로써 본 DSP에서 critical path가 형성되는 곱셈기의 속도 증가에 따라 사이클 수행에 걸리는 시간을 단축시킨다. 미국 VLSI사의 COMPASS 툴을 이용하여 검증된 바로는 fetch 사이클은 2ns이하에서 가능하며, 곱셈의 경우 데이타패스를 이용할 경우 지연 시간은 40ns 이하이다. 따라서 fetch/decoding 단계가 execution 단계에 비해 상대적으로 아주 짧기 때문에 execution 단계를 두 개로 나누어 파이프라인 단계를 줄임으로써 제어 구조를 단순화하였다. 그리고, FIFO 한 개의 명령어가 수행되는 사이클을 Fetch/Decoding, Read/Execution1, Execution2/Write의 3 사이클로 만들어 3 개의 명령어가 동시에 동작 가능하도록 설계되었다. 이러한 DSP를 개발하는 과정은 크게 4개의 부분으로 나눌 수가 있는데 먼저 전체적인 하드웨어 구조를 결정해야 하고 두 번째로 그 하드웨어 구조에 맞는 명령어 셀트를 결정해야 하며 세 번째로는 결정된 명령어 셀트에 맞는 어셈블러를 만들어야 한다. 마지막으로 어셈블리 코드에 따른 하드웨어의 동작을 보는 것이다.

본 논문의 구성은 2 장에서 DSP 의 하드웨어에 대한 설명을 하고, 3장에서 명령어 셀트의 종류 및 구성, 그리고 4 장에서 어셈블러에 대해 설명하고 있다.

본 DSP 구조는 다음과 같은 특징을 가지고 있다. 먼저 Dual port SRAM(Static RAM)과 shift register 기능을 동시에 갖는 FIFO(First In First Out)를 사용하고 있으며, 따라서 별도의 RAM과 PC(Program Counter)가 필요하지 않다. 또한 0~16 bit 까지 shift를 할 수 있는 barrel shifter를 사용하며, 14개의 범용 레지스터와 1개의 Accumulator를 사용한다.

Ⅱ. DSP 하드웨어 구성

FIFO를 사용하는 구조에서는 명령어를 FIFO에서 가져옴으로써 fetch에 걸리는 시간이 크게 줄어든다.[2][1] 따라서 fetch 사이클을 디코딩 사이클에 포함시켜 한 클럭에 수행할 수 있다. 마찬가지로 데이타 FIFO를 RAM처럼 사용하지 않을 때는 데이타 읽기와 쓰기에 걸리는 시간이 아주 작아진다. 따라서 읽기와 실행 1, 실행2와 쓰기가 동시에 가능하다. 그러나 FIFO를 RAM처럼 접속하는 몇몇 명령어들은 RAM 접속 시간을 무시할 수 없으므로 예외로 두고 처리해야 한다. 결과적으로 FIFO를 사용함으로써 명령어 파이프라인을 하지 않더라도 1 클럭에 1 명령어씩 수행할 수 있게 된다. 첫 번째 명령어 동작을 F1, 두 번째 명령어 동작을 F2, 세 번째 명령어 동작을 F3라 할 때 다음과 같은 시간 도표(timing diagram)를 얻을 수가 있다.

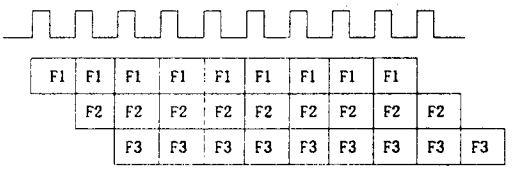


그림 1. 1클럭당 1명령어 수행 시간 도표
Fig. 1. Timing diagram for executing 1 instruction per 1 clock.

본 구조는 크게 2개의 부분으로 나눌 수가 있다. 첫째 메모리 부분, 그리고 두 번째로 연산 수행 부분이다. 다음은 각 부분의 동작 특성에 대한 설명이다.

1. 메모리 부분

메모리는 2개의 ROM(Read Only Memory)과 4개의 FIFO로 구성되어 있다. ROM은 각각 데이타 ROM과 명령어 ROM으로 구성이 되어 있는데 데이타 ROM에는 cosine table과 같이 연산에 필요한 기본적인 데이타들이 저장되어 있고 명령어 ROM에는 부팅이나 기타 시스템에 관계되는 인터럽트 벡터와 같은 내용이 들어 있다. 사용자가 작성한 프로그램이나 외부에서 입력되는 데이타등은 각각 명령어 FIFO와 데이타 FIFO에 저장이 된다. 4개의 FIFO 중 2개가 데이타 FIFO, 1개는 출력(output) FIFO, 나머지 1개는 명령어 FIFO이다. 특히 데이타 FIFO가 두 개 있는 이유는 다음과 같다. 이 구조는 DSP용으로 설계되었기 때문에 컨벌루션 동작이나 소프트웨어 필터링등을 수행할 때는 실시간 처리가 가능해야 한다. 따라서 한쪽에는 입력 데이타, 다른 한쪽에는 필터링 데이타를 저장하게 된다. FIFO를 제외한 FIFO들은 일반 FIFO와는 달리 RAM의 기능까지 해야 하므로 특수한 구조가 필요하다. FIFO의 구조는 Fig.2 와 같다. 각각 윗 부분이 RAM cell 부분, 아래쪽이 shifter 부분

으로 구성되어 있다. 데이타 버스는 2쌍이 있는데 이는 각각 읽기/쓰기 데이타 버스로서 FIFO에 읽고 쓰는 동작을 동시에 하기 위한 것이다. RAM 셀은 RD와 WR 신호에 따라 각각 데이타가 읽거나 씌여진다. Fig.2 에 대한 Pspice 가상 실험의 결과는 Fig.3 에 나타나 있다. 가상 실험 결과를 두 부분으로 나누어 놓았는데 a)가 SRAM part[3], b)가 shifter part이다. Fig.2 의 node 번호와 Fig.3 의 파형들을 비교해 보면 a)의 경우에 WR 이 high인 경우 데이타 버스에 실린 값이 RAM에 저장된 것을 알 수 있다. WR이 low인 경우 shift 신호가 발생하면 RAM에 저장된 데이타가 다음 셀로 이동을 하는데 b)를 보면 shift 신호가 rising edge일 때 RAM 셀에 저장된 데이타가 transmission gate M11을 통하여 node 102 까지 전달이 되고 다시 falling edge일 때 이 데이타가 M14를 통하여 node 103으로 옮겨감을 알 수 있다. 이러한 구조의 FIFO는 각각 명령어 FIFO와 데이타 FIFO에 사용된다.

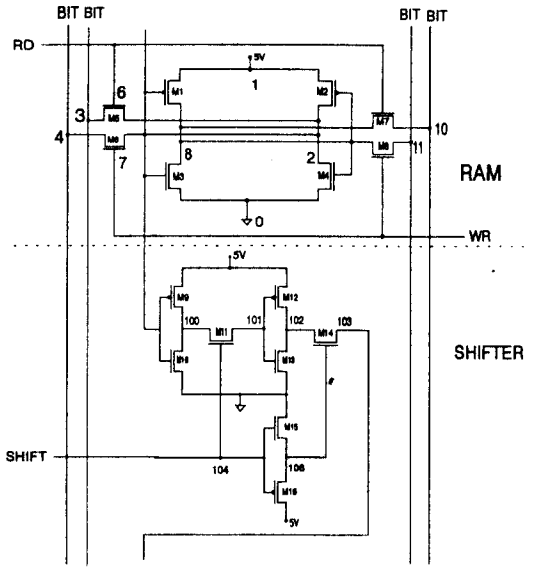
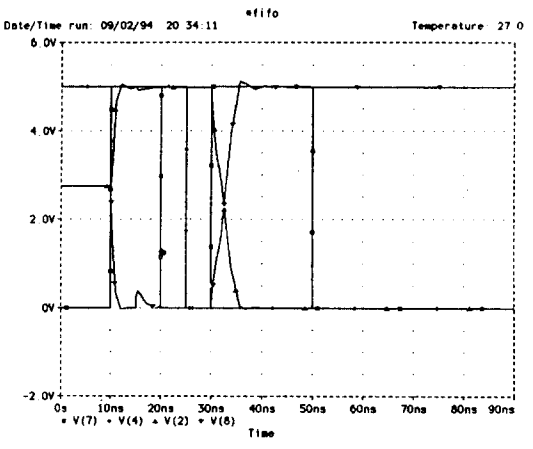
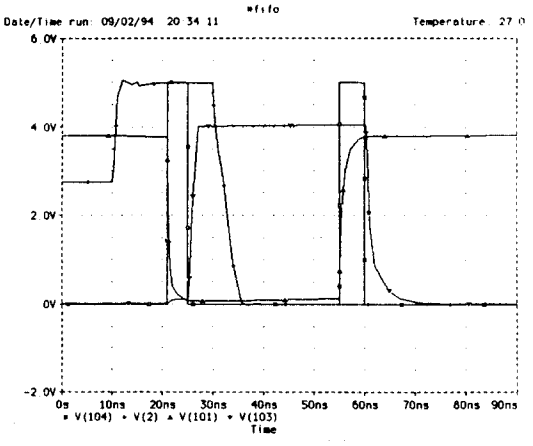


그림 2. FIFO 단위 셀 구조
Fig. 2. FIFO unit cell structure.

명령어 FIFO는 어셈블된 명령어 코드를 저장하는 곳으로서 shift 만으로 명령어를 내보내게 되므로 명령어 fetch 사이클에 소요되는 시간은 1회 데이타 shift 시간과 같다. 그러나 단순히 shift만을 사용하게 되면 FIFO내 명령어와 데이타를 update하는 overhead가 발생할 수 있다. 따라서 이를 극복할 수 있는 부가적인 제어 장치가 필요하다. 명령어 FIFO의 동작을 이해하기 위해 Fig.4 에 개략적인 구역 도표가 도시되어 있다.



(a)



(b)

그림 3. FIFO 단위 셀의 SPICE 시뮬레이션
Fig. 3. SPICE simulation of FIFO unit cell.

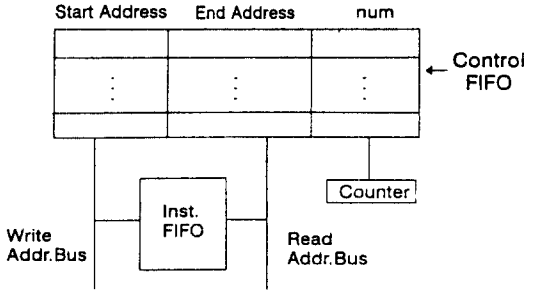


그림 4. 명령어 FIFO
Fig. 4. Instruction FIFO part.

제어 FIFO는 Start Address part, End Address part, 그리고 shift 횟수등 3 부분으로 구성되

어 있다.[4] 명령어 FIFO는 dual port SRAM의 기능 뿐만 아니라 일반적인 FIFO와 같이 데이타 shift의 기능까지 하게 되므로 읽기/쓰기 번지 버스에 번지 값을 주면 데이타가 FIFO의 시작 부분으로 들어가서 끝 부분으로 나오는 것이 아니고 write address 가 가리키는 곳으로 들어가서 read address가 가리키는 곳으로 나오게 된다. 즉, 명령어 FIFO 내의 데이타는 두 번지값이 가리키는 위치 사이에서 실질적으로 순환하게 된다. 이때 그 범위 밖에 저장된 명령어들은 아무런 이동이 없다. 만약 이런 기능이 없이 단순한 shift만을 이용해서 명령어를 수행하면 점프나 분기 명령을 사용할 수 없게 된다. 읽기/쓰기 번지 값은 제어 FIFO에 미리 저장이 되어 있다가 버스에 실리게 되는데 어셈블러에 의해 번지 값이 얻어진다. 어셈블러가 번지 값을 결정하는 기준은 다음과 같다. 예를 들어

```
LOOP:  LD    23          <--- Start Address
       MOV   R0   R14
       LD    10
       MOV   R5   R14
       ADD   R5   R0      <--- End Address
       JMP   LOOP
        .
        .
        .
```

이와 같은 어셈블리 코드가 있을 때 시작 번지와 끝 번지 사이의 블럭을 나누는 기준은 label과 Jump, Branch 명령어가 된다. 즉 lable이나 Jump, Branch 명령어 각각의 사이가 한개의 블럭으로 나뉘어져 위의 예와 같이 시작 번지와 끝 번지가 결정된다. 실제 read bus가 읽는 번지는 end address이므로 지정된 num 회수만큼 진행이 되지 않았다면 write address bus가 연결된 start address부터 다시 실행을 하게 된다. 이렇게 구분된 한개의 블럭이 FIFO 내에서 순환을 하면 전체 명령어가 흐트러지지 않고도 FIFO를 이용한 명령어의 수행이 가능하다. 이렇게 해서 한 개의 블럭에 대하여 shift가 모두 끝나면 제어 FIFO의 값이 shift되어 다음 수행될 블럭에 대한 번지가 버스에 실리게 된다. 제어 FIFO에 저장된 shift 횟수는 down counter로 로드가 되는데 카운터의 값이 0이 될 때 제어 FIFO가 shift 하게 된다. Shift 횟수는

$$num = Start\ Address - End\ Address + 1 \quad (1)$$

로 나타낼 수 있다.

데이타 FIFO 역시 같은 구조를 가지고 있는데 DSP data의 실시간 처리를 위해 외부로부터 연속적으로 흘러들어 오는 데이타를 위해 사용될 때는 일반적인 FIFO와 마찬가지로 shift 기능만을 하게 된다. 이는 플립플롭을 벡터 크기만큼 병렬로 이동시키는 것이기 때문에 PSPICE를 이용한 가상 실험이나 COMPASS를 이용한 실험 모두 2ns이하의 지연 동작을 한다. 반면에 데이타 FIFO에 데이타를 쓸 때는 RAM처럼 사용하게 되는데 이 때는 settling time 때문에 동작 속도에 제약을 받게 된다. 따라서 데이타 FIFO를 RAM처럼 사용해야 하는 몇몇 명령어를 제외하고는 RAM을 접속하는 경우가 없으므로 곱셈기 속도의 1/2인 20ns가 시스템의 사이클 속도가 된다. 따라서 약 50Mhz 정도의 시스템 속도를 얻는다. 출력 FIFO는 일반 FIFO와 같은 구조를 가지고 있는데 프로세서 외부로 데이타를 보낼 경우 일단 출력 FIFO에 저장한 후 한꺼번에 외부로 전송한다.

2. 데이타 연산 수행 부분

데이타 연산을 수행하는 부분은 크게 ALU, Multiplier, Barrel Shifter, Register등으로 나눌 수 있다.

이들 블럭들은 모두 16 비트 버스에 의해 서로 연결되어 있는데 입력 데이타는 FIFO, register, 외부 RAM으로부터 얻을 수 있다. 출력 데이타는 데이타 FIFO나 출력 FIFO에 저장되게 된다. 특히 최종 데이타를 레지스터가 아닌 데이타 FIFO에 저장하는 이유는 레지스터에만 데이타를 저장하게 되면 결국 데이타를 외부에서 가져와야 하기 때문에 데이타를 읽거나 fetch/디코딩 시간보다 길어지게 되어 critical timing path가 형성될 가능성이 높다. 또한 필터 데이타등을 사용할 수도 없게 된다. 각 연산의 사이클은 짧은 사이클 시간을 갖는 fetch 시간을 효율적으로 이용하기 위해 곱셈기의 연산 속도인 40ns을 1/2 파이프라인으로 나누어 두 사이클의 파이프라인을 형성한다. 전체 구역 도표는 Fig.5와 같다. 조건 플래그를 거친 후 디코딩된 어드레스에 해당하는 데이타가 데이타 FIFO에서 출력된 후 곱셈기나 ALU를 통과한다. 이때 연산기의 결과들은 데이타 필터링등의 동작이 필요할 경우는 다시 데이타 FIFO로 입력되며, 그렇치 않을 경우는 일차적으로 레지스터 블럭에 저장이 되며, CPU 밖으로 출력되기 위해 출력 FIFO에 저장된다. 레지스터 블럭의 출력이 다음 명령어 동작의 입력이 될 경우는 demux를 통해 연산기의 입력이 되는 경우도 있다. 그림에서 CONT.와 제어 FIFO를 다중화(Mux)해 놓은 것은 초기에 명령어를 IROM에서 가져올 때 CONT.에서 해당 위치를 지정하고 원하는 워드

만큼 명령어를 FIFO로 가져오기 위해서이다.

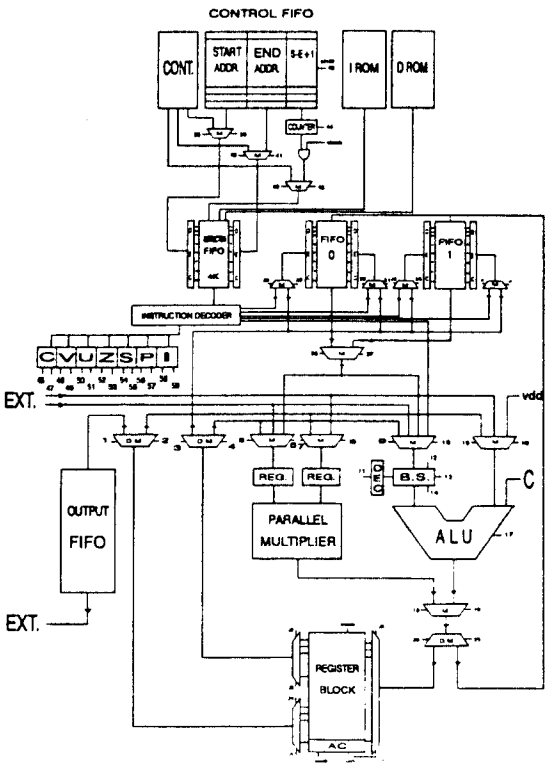


그림 5. 전체 블럭 선도
Fig. 5. Total block diagram.

Ⅲ. 명령어 셀트

본 구조에 대한 명령어 셀트를 결정함에 있어서 몇 가지 고려한 사항은, 첫째, 한 명령어 수행시 소요되는 클럭 수를 줄이기 위해 명령어 fetch는 한 번 되도록 하였으며, 둘째 데이타 FIFO가 RAM으로 사용될 수 있는 구조인 점을 고려하여 거기에 맞는 명령어를 선정 하였다. 셋째 데이타 FIFO를 RAM으로 사용하는 명령어와 그렇지 않은 명령어 사이의 동작 시간이 서로 다르다는 것을 구별하였다.

첫 번째 조건을 만족시키기 위하여 16 비트 내에 오피코드와 오퍼랜드를 모두 넣었다.[5] 따라서 명령어 종류와 오퍼랜드의 addressing mode에 제약이 있을 수 밖에 없다. 본 명령어 셀트에서 사용되는 addressing mode는 immediate mode, register mode, register indirect mode 그리고 FIFO 모드가 있다. FIFO 모드는 shift나 rotate 명령어에 적용될 수 있으며, 명령어, 데이타 FIFO와 함께 동작을 한다. 일반적인 산술 연산 명령어는 immediate 값이나 register형 오퍼랜드가 옴으로써 구조를 단순화하고 있다. 두 번째 조건을 만족하기 위해서 SETAI, SETAO, SETBI, SETBO등의 명령어를 추가하였다. 이들 명령어는 각각 2개의 데이타 FIFO에 대한 입력 번지와 출력 번지를 지정해 주는 것으로 이들 명령이 수행된 다음부터는 데이타 FIFO의 입출력 경로가 만들어지게 되어 이들 경로를 통하여 FIFO로 데이타를 보내게 된다. 따라서 FIFO에 데이타를 쓰기 위해서는 먼저 SET를 이용해서 경로를 만들어 준 후에 MOV등을 사용해서 데이타를 써야 한다. 일반적인 프로세서가 하나의 명령어로 RAM에 데이타를 쓸 수 있는 것과 비교하면 복잡한 과정을 거치게 되지만 제안한 DSP의 FIFO를 효율적으로 이용하기 위해서는 MOV등의 명령어는 2ns정도의 데이타 shift 시간이면 충분하므로 실제로 이 처럼 두 개의 명령어를 쓰더라도 일반 프로세서가 한 개의 명령어를 써서 RAM에 데이타를 보내는 것과 걸리는 시간은 별 차이가 없다. 물론 SET를 사용하지 않는 경우는 단지 mov명령만 수행되기 때문에 데이타 shift 동작만 이루어지므로 데이타 shift는 2ns정도면 동작이 가능하다. 마지막으로 세 번째 조건을 만족시키기 위해서 NOP 명령어를 두었다. RAM을 사용하는 명령어는 settling time 때문에 레지스터나 FIFO만을 사용하는 명령어 보다 수행 시간이 길게 마련이다. 그렇다고 해서 그러한 RAM 접속 명령어에 클럭 주기를 맞추면 RAM 접속 시간인 20ns 이하의 시스템 속도 향상을 기대하기는 어렵다. 따라서 RAM 접속 명령어를 수행할 때에는 다음에 NOP 명령을 삽입하여 RAM settling time동안 다른 명령어가 수행되지 않도록 하였다.

명령어 셀트를 이용하여 프로그램을 작성함에 있어 어셈블리어는 크게 명령어 부분과 오퍼랜드 부분으로 나누었다. 이러한 분류 기준을 따를 때 명령어를 오퍼랜드의 갯수에 따라 분류할 수가 있다.[6][7]

No operand의 경우는 STflag, CLflag, 그리고 조건 점프나 분기 명령등에서는 flag 레지스터의 내용이 관계가 있다. flag에는 Carry, Overflow, Underflow, Zero, Sign, Parity, Interrupt등 7개를 두었다.

STflag나 CLflag는 해당하는 flag를 1이나 0으로 만드는 명령인 반면 조건 점프나 분기는 flag 레지스터의 내용을 기준으로 조건을 판단하여 점프나 분기를 수행한다.

Ⅳ. 어셈블러 구성

어셈블리 프로그램의 한 라인은 다음과 같이 구성하였다.[8]

[label:] [Instruction] [operand 1] [operand 2] [;comment] (2)

물론 이중에서 label과 주석(comment)은 없어도 되는 부분이고 각 오퍼랜드의 갯수도 명령어의 종류에 따라 다르다. 어셈블러는 프로그램을 2번 scan 하여 기계어 코드를 생성하게 된다. 각각의 scan이 수행하는 동작은 다음과 같다.

SCAN 1에서는 먼저 각 label과 점프, 분기 명령어 간의 거리를 파악하여 저장하였다. 이는 점프나 분기 명령어로 인한 제어 FIFO동작이 원활하게 이루어지도록 하기 위한 조처이다. 그 다음 명령어의 사용이 올바른가를 검사하여 이상이 있을 경우에는 에러 메시지를 내보낸다. 마지막으로 각 오퍼랜드의 addressing mode 를 검색하여 scan 2 로 넘겨준다.

SCAN 2에서는 먼저 Scan 1으로부터 넘겨 받은 데이타를 이용하여 오피코드를 생성한다. 다음에 오피코드를 파일로 출력한다. 마지막으로 어셈블리 프로그램에서는 점프, 분기 명령을 위해 label을 사용할 수가 있는데 label은 ':'에 의해 구분이 된다.

또한 프로그램내에서 주석을 달 경우는 ';' 으로 시작되는 문장을 어셈블러가 주석으로 인식하여 어셈블리 코드를 만들지 않는다. ':', ';'는 모두 첫 번째 scan 할 때 처리되는 부분이다.

V. 결 론

본 논문에서는 새로운 구조의 DSP chip에 대한 연구를 수행하였다. 특별한 용도의 DSP 프로세서를 개발하기 위해 하드웨어를 설계하면 물론 속도에는 잇점이 있지만 모든 용도에 대해서 각각의 하드웨어를 설계하기란 매우 힘들다. 따라서 각 알고리즘에 대한 프로그래밍만으로 이러한 하드웨어를 구현하기 위해서는 매우 빠른 DSP 프로세서가 있어야만 좋은 결과를 얻을 수 있다. 이를 위하여 FIFO로부터 명령어를 fetch 하는 새로운 DSP 구조를 고안했는데 RAM 으로부터 데이타를 fetch해 오는데 걸리는 RAM 접속 시간과 FIFO의 이동 시간을 비교해 볼 때 이러한 구조를 사용함으로써 50Mhz 동작 속도까지 얻을 수 있다. 또한 번지를 계산하는 ALU가 필요 없게 되어 구조가 더욱 간단해지고 버스 숫자가 줄어들게 되어 칩 면적에도 많은 부분 감소가 있게 된다. 이 구조는 범용 DSP에 촛점이 맞추어져 있기 때문에 전용 DSP에 응용할 경우 다소의 구조 변경을 통해 응용할 수 있을 것이다.

참 고 문 헌

[1] A.W.M.Van Den Enden, N.A.M. Verhoecky, *Discrete-time Signal Processing*, Prentice-Hall,Inc. pp. 4-15, 1989.

[2] Sajjan G.Shiva, *Computer Design & Architecture*, Harper Collins, pp. 147-150, 1991.

[3] Neil Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, A.W., pp. 348-354, 1985.

[4] Fredrick J.Hill, *Digital Systems*, WILLY, pp. 283-286 1987.

[5] *TMS320C3x User's Guide*, TI, Inc. 1991.

[6] *WE DSP32C Digital Signal Processor Information Manual*, AT&T,Inc. 1990.

[7] *WE DSP16 and DSP16A Digital Signal Processor Information Manual*, AT&T, Inc. 1989.

[8] Herbert Schildt, *C++*, McGraw Hill, 1992.

저 자 소 개

金 榮 民(正會員)

1954년 4월 18일생. 1976년 2월 서울대학교 전자공학과 졸업(공학사). 1978년 2월 한국과학원 전기및전자공학과 졸업(공학석사). 1978년 3월 ~ 1979년 7월 한국선박해양 연구소(주임연구원). 1979년 8월 ~ 1982년 7월 국방과학연구소(연구원). 1986년 오하이오 주립대학교 전기공학과(공학박사). 1988년 6월 ~ 1991년 8월 한국전자통신연구소(실장). 1991년 9월 ~ 현재 전남대학교 전자공학과 교수, 주요 관심분야는 영상압축, VLSI 설계, 신경회로망등임.

朴 柱 炫(正會員)

1969년 7월 13일생. 1993년 2월 전남대학교 전자공학과(공학사). 1995년 2월 전남대학교 전자공학과 대학원 졸업(공학석사). 1995년 3월~ 현재 전남대학교 전자공학과 대학원 박사과정. 주요 관심분야는 VLSI 설계, 음성 압축등임.