마이크로 컴퓨터 특집을 내면서

김 창 은
(금성사 중앙연구소 정보기기 연구소 부소장)

1977년 COMMODORE에서 PET, TANDY에서 TRS-80등의 8bit 마이크로 프로세서를 이용한 마이크로 컴퓨터 출범이래 11년이 지난 현재, 64bit RISC (Reduced Instruction Set Computer)를 이용한 SUPER PERSONAL COMPUTER (APOLLO)가 등장하여 60 MIPS (Million Instructions Per Second) 이상의 성능을 가지는 컴퓨터로 성장하고 있읍니다. 마이크로 컴퓨터를 정확히 정의하고 구분하기는 힘들지만, 일반적으로 주 전산 기능장치 MPU, 주기억장치 Memory, 주변기기와의 Interface 장치 및 전반적 System 관련기능의 OS (OPERATING SYSTEM)가 중요한 필요기능 입니다.

MPU 기능을 위한 연구개발은 8bit 마이크로 프로세스(CISC Architecture)에서 64bit RISC로, Single Processor에서 Multiple, Parallel 또는 Distributed Processor System으로 발전되고 있읍니다. 본 특집에서는 RISC, Transputer 및 Neural NET Architecture의 원고에서 그 기술 발전 추이를 검토하여 보았읍니다. 반도체의 Submicron 기초기술과 사용자의 응용 소프트웨어에서의 높은 성능 요구로 인하여 기능 집적이 계속되고, 특별기능 전용 CPU (즉 Embedded CPU)가 계속 연구개발될 전망 입니다.

Memory 분야도 접적화가 계속되어, 16/64 Mega-bit DRAM이 개발되고 있고, Multiport Memory의 이용으로 Bandwidth를 늘여 High Speed CPU의 성능을 충분히 이용하고 있읍니다. 앞으로 Memory의 수요는 기하급수적으로 늘어 날것이며, CPU의 고속기능을 발휘시키기 위한 Cache Memory Architecture 또는 새로운 방법(Multiport, Physical Page 등)의 연구가 활성화 하며 Memory의 효율성을 높여야 할것 입니다.

Input/Output Interface 분야 또한 특수응용목적용으로 집적화(ASIC)가 되고 있고 Interface의 표준화는 더욱 그 속도를 증가 할 것입니다. Display 및 Keyboard등을 이용한 Visual Man-Machine Interface는 많은 발전을 보았으나, 인간에게 친근하기 위한 Aural/Speech Interface가 절대 필요하고 연구개발을 요하는 분야 입니다.

마이크로 컴퓨터의 자원을 최적으로 관리 운영하고 사용자와 기초적 연결을 시켜주는 OS 분야도 발전하였으나 표준화, 일반화 등의 제약조건으로 인하여 다른 분야의 발전속도에는 못 미쳤다고보겠읍니다. 본 특집에서는 현 OS의 동향을 전반적으로 소개하고 일본의 TRON 및 그에 맞는 Micro Processor 응용분야, 중요한 데이타 베이스와 연결되는 OS (본고에서는 Main Memory 데이타 베이스로 함)를 실어 보았읍니다. 이 분야도 MPU와 Interface에서 처럼 특정응용 목적용 OS, 즉 ASOS (Application Specific Operating System)의 발전을 예상하고 있읍니다.

전반적인 마이크로 컴퓨터의 산업동향으로 볼때 기술발전 및 경제성에 힘입어 분야별 목적별로 A-SSP (Application Specific System Product)화 되면서 Human Interface, Network 및 Communication이 필요조건으로 부수되겠읍니다. 연구개발 및 응용분야가 광범위한 마이크로 컴퓨터 전반을 다루는 것이 무리이지만, 본 특집이 또하나의 연구개발 활성화의 기회가 되기를 바라며 원고를 내주신 필자 여러분께 감사 드립니다.

특집 순서

로세서에게 할당할 수 있다면 코프로세서의 사용이 적합할 것이다.

Flash Multiplier의 속도는 IMS T800의 입장에서는 필요 이상으로 빠르다. 파이프라인 벡터 프로세서(pipelined vector processor) 같은 것만이 Flash Multiplier의 처리 속도에 맞추어 operand를 전달할 수 있다.

배럴 쉬프터(Barrel shifter)도 실리콘 면적을 많이 차지하므로 적합하지 않다.

결국, 크기가 작은 고속 정규화 쉬프터(Fast Normalizing shifter)가 사용되었다. IMS T800에서 이것은 쉬프트에 1 cycle, 정규화에 2 cycle이 걸린다. 이 정규화 쉬프터에 곱셈과 나눗셈을 위한 기타 논리 회로가 부가되어 FPU를 이룬다.

이때 한 cycle에 곱셈은 3비트, 나눗셈은 2비트씩 결과가 나온다. 이로써 Single length의 곱셈에 13 cycle(T800-30은 433 ns, T800-20은 650 ns) 걸린다. Double length의 나눗셈은 34 cycle(T800-30은 1130 ns, T800-20은 1700 ns)이 걸린다.

2.3.2 표준함수의 계산

이제 부동점 산술 연산에 필요한 sqrt(제곱근)이나 sin 같은 표준 함수의 구현을 생각해 보자. IMS T800은 두 가지 근사법(Approximation)을 비교해 보았다.

첫번째로 Cordic 방법은 하드웨어 구현을 위해 고안되었으므로 빠르긴 하지만 데이터 경로(data path)에 많은 하드웨어를 필요로 하며 큰 탐색표(look-up table)가 필요하다. 이 하드웨어로 최고 성능이 4 cycle마다 1비트 결과를 생성할 수 있다면 최소 계산시간은 double length에 대해 230 cycle이 필요하다.

Intel 8087 코프로세서에서 Cordic 방법을 사용하고 있다.

두번째 방법은 다항식 근사법(polynomial approximation) 으로 T800에서 사용했다. 이것은 FPU안에 부가적인 하드웨어를 전혀 요구하지 않는다. 함수마다 다소 다르지만 sin 함수의 계산은 Cordic 방법의 2배 밖에 걸리지 않는다. 이것은 곱셈 구현이 한 cycle당 3비트씩 결과를 낼 수 있기 때문이다.

사실상 우린 계산 시간만을 고려했고 인수 축소(argument reduction)와 함수 생성(function generation) 시간은 고려하지 않는다. 이들을 고려하면 Cordic 방법이 polynomial approximation보다 2배 빠르다는 것은 거짓이다. T800은 삼각함수 계산을 위해 보조 하드웨어를 포함하고 있지 않다.

제곱근 계산을 위해서는 약간의 하드웨어가 필요했다. IEEE 표준이 최종 비트까지 정확할 것을 요구하는데 단순한 다항식 계산으로는 어렵기 때문이다. 그러나 필요한 하드웨어의 양은 대단치 않다.

그림 9의 블록도에서 data path는 쉬프트 경로를 포함한다. 가수(Fraction)부의 데이터 경로는 59 비트이고 지수(Expoent)부는 13 비트이다. 정규화 쉬프터는 가수부의 데이터를 지수부의 값에 의해 적당한 양만큼 쉬프트하여 인터페이스를 통해 출력한다.

그림상엔 ROM 이 두개로 표시 되어 있지만 사실은 하나의 ROM이다. 이것은 ROM이 논리적으로 두 부분으로 분할되어, 제어신호들이 버스를 쓰지 않고 직접 전달됨을 뜻한다.

2.4 부동점 연산 성능(Floating Point performance)

IMS T414의 마이크로 코드는 32비트 부동점 연산을 지원해 현재 부동점 연산용 코프로세서 정도의 성능을 내었다. Single length IEEE 754 부동점수에 대해 약 10 마이크로초의 연산 시간을 보였다. IMS T800-20은 FPU를 첨가함으로써 T414보다 10배 이상 시간을 단축했다. (표 4)

연산시간은 실제 프로그램의 성능 척도로 적합치 않으므로 Whetstone 벤치마크가 사용되었다. 이는 부동점 연산을 적절히 혼합하고, 프로시듀어 호출과 어레이 인덱싱(array indexing) 및 고급 함수를 포함한다.

(표 5)를 보면 T414가 MC 68881 코프로세서보다 연산시간이 세배나 더 걸리지만 벤치마크에 의한 비교로는 25% 정도밖에 성능이 뒤지지 않음을 볼 수 있다. 이는 부

표4 부동점 연산시간

	IMS T800-30		IMS T800-20		IMS T414-20	
Operation	Single	Double	Single	Double	Single	Double
Add	233 ns	233 ns	350 ns	350 ns	11.5 μs	28.3 μs
Subtract	233 ns	233 ns	350 ns	350 ns	11.5 μs	28.3 μs
Multiply	433 ns	700 ns	650 ns	1050 ns	10.0 μs	38.0 μs
Divide	633 ns	1133 ns	950 ns	1170 ns	12.3 μs	55.75 μs

표5 Whetstone Benchmark에 의한 비교

Processor	Type	Whets / s single length
Intel 80286 / 80287	8 MHz	300K
IMS T414-20	20 MHz	663K
NS 32332-32081	15MHz	728K
MC 68020 / 68881	16 / 12 MHz Sun3	860K
VAX 11 / 780 FPA	Unix 4.3 BSD	1083K
IMS T800-20	20 MHz	4000K
IMS T800-30	30 MHz	6000K

동점 수식의 계산이 계산속도 뿐만 아니라 데이터의 이동 시간에도 영향을 받기 때문이다. 이런 요인들을 잘 balancing 함으로써 단일 칩인 IMS T800-30은 Intel 80386 / WTL 1167 코프로세서 칩세트의 조합과 비슷한 면적으로 이보다 6배의 성능을 내고 있다. 게다가 IMS T800은 외부 메모리 없이도 쓸 수 있고 보드상에 시스템을 구현할 때 필요한 보조 회로의 양이 적다.

2.5 Formal specification language

2.5.1 정확하고 빠른 디자인을 위한 시스템 기술 언어

마이크로 프로세서로 life-critical한 시스템을 디자인할때 정확해야 할 뿐 아니라 시장성을 고려해야 하므로 빠른 디자인이 요구된다. 복잡한 시스템을 모든 조건에 대해 테스트하는 Exhaustive test는 바람직하지 않다.

Inmos 회사는 ANSI / IEEE 표준 754(1985년 Binary Floating point Arithmatic을 위해 제정)의 정확한 구현을 위해 매우 진보된 정형법(Formal method)을 사용했다. 이는 옥스포드 대학의 프로그래밍 연구 그룹(Programming Research Group)과 협력해 개발되었고 Occam의 formal semantics를 이용했다.

Inmos는 복잡한 디자인에 formal method(공식화된 방법)을 사용함으로써 정확성을 기할뿐 아니라 디자인 시간이 감소된다는 사실을 발견했다.

시스템 구현시 먼저 Occam으로 소프트웨어 패키지를 짜서 시스템 명세(specification)에 맞는지 확인후 IMS T800 마이크로 코드로 변환하게 된다.(그림 10)

2.5.2 Formal specification language Z

Jean-Raymond Abrial로부터 유래해 Oxford 대학의 프로그래밍 연구 그룹에 의해 개발 사용되어 온 spec notation Z는 실제 시스템의 동작을 정확하고 쉽게 기술하기 위해 사용된다.

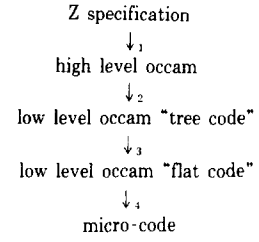


그림10 Occam과 Z 및 마이크로 코드의 관계

Z 명세는 formal text와 자연언어 설명(natural language description)으로 구성된다. 다시 formal text는 스키마 언어(schema language)와 수학적 언어(mathematical language) 로 나뉜다. 전자는 구조화를 위해 후자는 추상적인 수학적 표현을 위한 것이다.

Count

value, value' : N reset? : bool
reset? ⇒ value' = 0 ¬reset? ⇒ value' = value + 1

그림11 Z 명세의 예 1

(그림 11)를 보면 스키마는 두 개의 box로 나뉜다. 위의 box는 변수 선언, 아래의 box는 변수가 가질 값들을 표시하는 서술부(predicative)들로 구성된다.

이때 unprimed variable(valve')은 primed varable(-valve) 의 계산 전의 값을 의미한다. 위의 예는 입력이 reset이면 valve를 0으로 아닐 때는 valve를 1씩 증가하는 시스템을 기술하고 있다.

(그림 12)의 Z 명세는 Areg 레지스터 내의 부동점수가 정수범위 MinInt와 MaxInt 사이의 범위에 포함되는지 검사해 애러 플래그를 서트하는 것이다.

이것으로 (그림 10)의 각 레벨 예를 (그림 12)에서 (그림 15-1)에 보인다. 자세한 설명은 생략한다.

Floating_Check_Integer_Range

Areg, Areg' : Floating_Point_Register Error_Flag, Error_Flag' : bool
fv ∈ Z Areg' = Areg fv Areg ∈ [MinInt, MaxInt] ⇒ Error_Flag' = Error_Flag fv Areg ∉ [MinInt, MaxInt] ⇒ Error_Flag' = true

그림12 Z 명세의 예 2

```
IF
  Areg. Exp < LargestINTExp
    SKIP
  (Areg. Sign = NEGATIVE) AND
   (Areg. Exp =LargestINTExp) AND
                (Areg. Frac = MSBit)
    SKIP
  TRUE
    SetError (ErrorFlag)
```

그림13 high level Occam 구현

```
SEQ
  AregSignNEGATIVE := (Areg. Sign = NEGATIVE)
  ExpZbus := (Areg. Exp - LargestINTExp)
  ExpZbusNeg := ExpZbus < 0
  IF
    AregSignNEGATIVE
      ··· negative case
    NOT AregSignNEGATIVE
      IF
        ExpZbusNeg
          SKIP
        NOT ExpZbusNeg
          SetError (ErrorFlag)
```

그림14 low level Occam "tree code"

```
INT NextInst :
SEQ
  NextInst := FloatingPointCheckIntegerRange
  WHILE NextInst <> NOWHERE
    IF
      NextInst = FloatingPointCheckIntegerRange
        SEQ
          Areg Sign NEGATIVE := (Areg. Sign =
NEGATIVE)          ExpZbus := (Areg. Exp - Lar-
gestINTExp)          ExpZbusNeg := ExpZbus < 0
          IF
            AregSignNEGATIVE
              ··· negative case
            NOT AregSignNEGATIVE
              IF
                ExpZbusNeg
                  NextInst : NOWHERE
                NOT ExpZbusNeg
                  NextInst := OutofRange
      NextInst = OutofRange
        SEQ
          SetError (ErrorFlag)
          NextInst := NOWHERE
      ··· negative case micro instructions
```

SetError (ErrorFlag) process moced into a separate microin struction.

그림 15 low level Occam "flat code"

```
Floating Point Check IntegerRange :
ConstantLargestINTExp
ExpXbusFromAreg ExpYbusFromConstant
ExpZbus FromXbus MinusYbus
GOTO Condl FromAregSing - >
      (Cond0From ExpZbusNeg - > (···, ···),
      Cond0From ExpZbusNeg - >
      (NOWHERE, OutofRange))
```

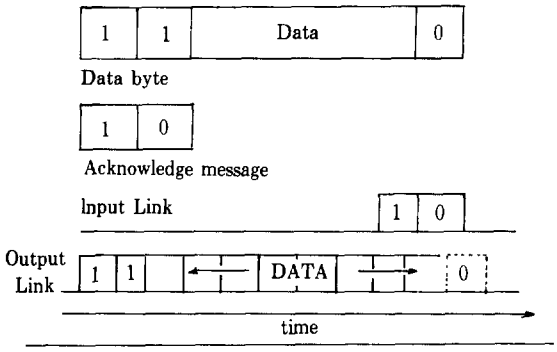
그림 15-1 그림 15 6째 줄 Floating Point Check Integer Range 해당 마이크로 코드

소프트웨어 패키지는 먼저 Occam으로 구현해 Z 명세로 번역해 정확성을 검사한다. 이런 변환은 ML언어로 짜여진 Occam source transformation system을 이용한다. 정확성이 증명되면 Interactive program transformation system을 이용해 마이크로 코드로 번역한다.

2. 6 통신 링크(Communication link)

두 트랜스퓨터는 두 개의 단방향 회선으로 링크 인터페이스끼리 연결하므로써 지점간 직접 전송한다. 이 두 선은 두 개의 Occam 채널에 해당한다. 한 선으로 데이터를 보내고 다른 선으로 애크노리지(Acknowledge : 이후 ACK로 표시함) 신호를 받게 된다. 통신을 위한 프로토콜(protocol) 은 (그림 16)처럼 매우 간단하다.

바이트 단위로 통신이 이루어진다. 시작 비트(start bit' : 값 1)후에 1을 한번 더 보내고 1 바이트의 데이터를 보낸후 중지 비트(stop bit : 값 0)를 보낸다. ACK 신호를 받아야 다음 데이터를 보낼 수 있다. ACK는 시작 비트 1과 중지 비트 0만으로 되어 있다.



Signals when acknowledge packet overlaps data packet.

그림16 통신 프로토콜

IMS T414는 데이터를 다 받은 후에야 ACK를 보내지만, T800은 데이터 패킷(packet)을 인식하자 마자 끝까지 기다리지 않고 바로 ACK를 보낸다. 데이터 패킷과 ACK 패킷의 이런 오버랩에 의해 IMS T414의 0.8 Mbyte/ s의 전송률을 1.8 Bbyte/s로 향상시켰다. 양방향 통신의 경우 2.4 Mbyte/s의 전송률을 보였다.

2.7 그래픽 능력(Graphic capability)

비트 단위의 그래픽 프로세서는 오늘날 픽셀당 1바이트의 컬러 코드를 주는 디스플레이 장치(byte-per-pixel color display)에 적합치 않다. IMS T414의 빠른 블럭 전송(block-move)은 이런 디스프레이 장치를 사용하는 그래픽에 적합하다. IMS B007 컬러 그래픽 보드(IMS B007 color graphics evaluation board)에 사용되었다.

IMS T414의 블럭 전송은 메모리 대역폭을 최대로 활용하기 위해 디자인했다. 메모리의 임의의 블럭을 임의의 위치로 옮기는데 read와 write 회수를 최소화 했다.

트랜퓨터의 내부 메모리에서 약 60 Mbytes/s (T414-30), 40 Mbytes/s (T414-20)의 전송률을 보인다. 가능한 가장 빠른 메모리를 외부 메모리로 연결했을때 20 Mbytes/ s (-30), 13.3 Mbytes/s (-20)의 전송률을 가진다.

2.7.1 Move2d, Draw2d, and Clip2d

IMS T800은 이런 블럭 전송을 2차원적으로 확장했다. 메모리 대역폭을 최대로 활용하면서 윈도우(window)를 화면의 어디에나 옮길 수 있다.(그림 17. Move2d)

템프레이트(template)나 텍스트를 윈도우 안으로 옮기기 위해, 픽셀값을 테스트해 픽셀값이 0이 아닌 것만(그림 18. Draw2d), 또는 0인 것만(그림 19. Clip2d) 전송하는 조건부 블럭 전송도 만들어졌다.

이들 모두는 T414의 단순한 블럭 전송과 비슷한 속도를 얻을 수 있다. 1 백만 픽셀의 스크린을 초당 13번 그릴 수 있다.

```
PROC Move2d ([] []BYTE Source, sx, sy,
             [] []BYTE Dest, dx, dy,
             width, length)
  SEQ y = 0 FOR length
    [Dest [y+dy] FROM dx FOR width] :=
      [Source [y+sy] FROM sx FOR width]
```

그림17 Source[sy][sx]에서 Dest[dy][dx]로 픽셀값을 복사(Move2d)

```
PROC Draw2d ([] []BYTE Source, sx, sy,
             [] []BYTE Dest, dx, dy,
             width, length)
  BYTE temp :
  SEQ line = 0 FOR length
    _SEQ point = 0 FOR width
      SEQ
        temp := Source[line+sy] [point+sx]
        IF
          temp = 0(BYTE)
            SKIP
          TRUE
            Dest[line+dy] [point+dx] := temp
```

그림18 값이 0이 아닌 것만 복사(Draw2d)

```
PROC Clip2d([] []BYTE Source, sx, sy,
            [] []BYTE Dest, dx, dy,
            width, length)
  BYTE temp :
  SEQ line = 0 FOR length
    SEQ point = 0 FOR width
      SEQ
        temp := Source[line+sy] [point+sx]
        IF
          temp = 0(BYTE)
            Dest[line+dy] [point+dx] := temp
          TRUE
            SKIP
```

그림 19 값이 0인 것만 복사(Clip 2d)