論文98-35C-12-8

가변 길이 명령어 처리를 위한 명령어 버퍼 구현

(Implementation of an Instruction Buffer to process Variable-Length Instructions)

朴 柱 炫 *, 金 榮 民 *

(Ju-Hyun Park and Young-Min Kim)

요 약

본 논문에서는 명령어 버퍼에 저장되어 있는 가변 길이 명령어의 미스 율을 낮추기 위한 버퍼를 구현한다. 또한 반복적으로 수행되는 명령어들의 디코딩 시간을 줄이기 위해 외부에서 명령어를 패취(fetch)하여 초기 디코드 동작을 한 후 그 결과를 버퍼에 저장하는 MAU(Mark Appending Unit)를 둔다. 또한 분기 명령어의 효과적인 처리를 위해 타겟 명령어의 히트 여부를 판단하는 회로를 내장하고 있다. 가변 길이 명령어를 처리하기 위한 각 모듈은 VHDL을 이용해 설계되었으며, Model Technology Inc.의 V-System를 사용하여 시뮬레이션 하였다. 합성 및 검증은 0.6㎛ 5-Volt CMOS TLM(Three Layer Metal) COMPASS 라이브러리를 이용한 ASIC Synthesizer 툴을 사용하였다. 최고 동작 속도는 약 140MHz까지이며, 총 게이트 수는 약 17,000개이다.

Abstract

In this paper, we implement a buffer capable of handling short loops references to statistically lower the miss rate of variable-length instructions stored in the instruction buffer. MAU(Mark Appending Unit) takes the instructions as they are fetched from external memory, performs some initial decode operations and stores the results of the decode in the buffer for reducing multiple decodes when instructions are executed repeatedly such as in a loop. It includes a decision block of whether hit or not for effectively processing branch instructions Each module of the proposed architecture of processing variable-length instruction is described in VHDL structurally and behaviorally and whether it is working well or not is checked on V-System simulator of Model Technology Inc. We synthesized and simulated the architecture using an ASIC Synthesizer tool with 0.6㎛ 5-Volt CMOS COMPASS library. Operation speed is up to 140MHz. The architecture includes about 17,000 gates.

Ⅰ. 서 론

최근에 프로세서는 반도체 집적 기술과 동작 속도 개선으로 그 성능이 크게 향상되고 있다. 그러나 더 높은 성능 향상을 요구하는 분야가 계속 발생하고 있다. 이러한 요구를 만족하기 위하여 RISC(Reduced Instruction Set Computer)라는 구조가 제안되었다. RISC는 보통 고정 길이 명령어 코드를 가지고 있으며, 산술 및 논리 연산 동작은 단지 레지스터 사이에서 이행이 된다. 뿐만 아니라 메모리 접근은 단지 메모리와 레지스터 사이의 데이터 전송만을 허용한다. 이와 같은 제한 조건으로 명령어 수행에 필요한 하드웨어는 단순화되고, 하드웨어 제어도 간단해진다. 따라

* 正會員, 全南大學校 電子工學科
(Dept. of Electronics, Chonnam National Univ.)
接受日字:1998年6月29日, 수정완료일:1998年11月19日

서 프로세서의 동작 주파수를 높이고, 한 명령어를 수행하는데 필요한 사이클 수를 줄임으로써 성능을 향상시킬 수 있는 장점이 있다[1] [2]. 그러나 네트워크 응용의 발전에 따라 서브루틴 호출 동작이 많은 객체 지향형 프로그램이 많이 사용되고 있으며, 이와 같은 프로그램을 수행할 때의 효율적인 구조는 스택을 기반으로 한 프로세서인 것으로 알려져 있다[3]. 그러나 스택 기반 명령어들은 오퍼랜드를 나타낼 필요가 없기 때문에 레지스터 기반 명령어와는 달리 짧은 명령어 길이를 가지며, 반면에 영상 데이터를 처리하는 레지스터 기반의 응용에서는 레지스터 필드를 나타내야 하기 때문에 스택 기반 명령어보다 긴 명령어 길이를 갖는다. 따라서 영상 데이터를 네트워크 기반의 객체 지향형 프로그램으로 처리하기 위해서는 레지스터를 기반으로 하는 명령어와 스택을 기반으로 하는 명령어의 공존이 필요하다.

명령어 길이가 길어질수록 더 긴 오피코드와 오퍼랜드를 이용하여 보다 짧은 프로그램으로도 같은 일을 수행하는 프로그램 작성이 가능하며, 융통성있는 기능을 구현할 수 있도록 다양한 어드레싱 모드를 지원할 수 있다[4]. 그러나 명령어 길이가 길어질수록 낭비되는 측면이 있다. 32비트 명령어가 16비트 명령어에 비해 항상 2배의 유용성을 갖는 것은 아니다. 따라서 명령어 길이 증가로 인한 메모리 데이터 전송율이 CPU 속도 증가에 병목 요인이 되지 않도록 하기 위해 명령어 캐쉬, 명령어 버퍼를 사용하거나 명령어 길이를 보다 짧게 만든다. 대부분의 RISC는 명령어 캐쉬와 명령어 버퍼를 동시에 사용하고 있으며, 분기 동작에 대한 예측을 하는 전용 블록을 가지고 있다. 따라서 명령어 버퍼는 단순히 우선패취(prefetch) 버퍼로만 사용이 되며, FIFO(First In First Out) 동작을 하는 몇 개의 레지스터로 구현된다[5] [6] [7] [8] [9].

그러나 RISC는 고속 처리를 위해 명령어가 단순하고, 고정 길이를 가지고 있기 때문에 스택을 이용한 명령어에는 낭비되는 측면이 있다. 따라서 레지스터와 스택을 동시에 사용하는 프로세서는 고정 길이 명령어를 이용하는 것보다 가변 길이 명령어를 이용하는 것이 많은 이점이 있다. 가변 길이 명령어는 고정 길이 명령어보다 메모리 크기를 절약할 수 있으며, 같은 성능을 갖는 고정 길이 구조보다 더 작은 버퍼를 필요로 한다. 또한 버스 대역폭(bandwidth)을 줄일 수 있어 적은 비용의 메모리를 사용할 수 있다[10]. 그러나 가변 길이 명령어는 CPU의 복잡도를 증가시키는 문제점을 갖는다. CPU는 다음 명령어 길이를 알지 못하기 때문에 가장 긴 명령어 길이 단위로 패취할 필요가 있으며, 이것은 때때로 다중 명령어를 패취할 수 있음을 의미한다[11]. 뿐만 아니라 명령어 길이 정보를 디코딩 단계에서 이용할 수 있는 우선 디코드(predecode) 단계가 포함되어야 한다.

우선 디코드는 메모리에서 명령어를 패취해서 초기 디코드 동작을 한 후 그 결과를 내부 버퍼에 명령어 길이 정보와 함께 저장한다. 이런 정보에는 명령어 길이 정보 뿐만 아니라 명령어가 어떤 수행 유닛을 사용하는지 여부를 판단할 수 있는 제어 신호도 포함된다. 우선 디코드의 이점은 명령어 디코더 로직의 복잡도를 줄임으로써 디코딩 동작을 빠르게 할 수 있으며, 근거리 루프처럼 연속적인 명령어들을 반복적으로 수행할 때의 다중 디코딩 동작을 줄인다[2] [6] [9].

본 논문에서는 외부 메모리로부터 패취한 가변 길이 명령어를 저장하기 위한 명령어 버퍼를 설계한다. 명령어 버퍼는 데이터 처리를 위한 제어 신호를 발생할 수 있도록 디코더에 명령어와 명령어 길이 정보를 함께 보낸다. 또한 분기 동작이 일어날 때 명령어 버퍼 내에서 우선적으로 분기 타겟 명령어를 찾을 수 있도록 하는 제어 블록을 가지고 있다.

본 논문의 구성은 Ⅱ장에서는 명령어 버퍼 구조를 기술한다. Ⅲ장에서는 VHDL을 이용한 설계 및 합성을 기술하고, Ⅳ장에서는 시뮬레이션을 통해 확인된 결과를 분석한다. 마지막으로 Ⅴ장에서는 본 연구의 결과를 기술하고 향후 연구방향을 제시한다.

Ⅱ. 명령어 버퍼 구조

그림 1은 명령어 길이 정보를 포함하는 명령어 버퍼 구조이다.

그림 1은 외부 프로그램 메모리에서 32비트 길이의 명령어를 입력받아 디코더 블록으로 명령어 길이 정보와 함께 보내는 명령어 버퍼이다. 이 버퍼는 가변 길이 명령어를 처리하기 위한 구조로 프로그램 메모리로부터 입력된 명령어의 길이 정보를 디코딩하기 위한 MAU(Mark Appending Unit)을 내장한다. 또한 가변 길이 명령어를 저장하는 IBU(Instruction Buffer Unit)와 MAU로부터 출력된 명령어 길이 정보를 저장하는 MB(Mark Buffer), MB 정보가 의미가 있는

지 여부를 판단해 주는 FB(Flag Buffer)를 내장한다. 다음 패취될 명령어가 버퍼에 존재하면 상응하는 가변 길이 명령어와 그 길이 정보는 각각 IBU, MB로부터 CPU에 전달된다. 반면에 다음 패취될 명령어가 버퍼 안에 존재하지 않으면, 상응하는 가변 길이 명령어는 외부 프로그램 메모리로부터 IBU로 다시 패취하게 되며, 명령어 길이는 MAU에서 다시 디코딩된 후 길이 정보와 함께 CPU에 전달된다. 따라서 이때의 가변 길이 명령어가 CPU에서 디코딩될 때까지의 시간은 IBU에 다음 수행될 명령어가 존재할 때보다 더 길게 걸린다. MAU는 명령어 우선디코드 특성을 갖는다. 즉 명령어가 디코딩 되기 전에 가변 길이 명령어를 디코딩한 후 길이 정보를 출력하기 때문이다. IBCU (Instruction Buffer Control Unit)는 명령어와 길이 정보를 읽거나 쓸 때 버퍼내의 명령어 위치를 가리키는 포인터를 발생시켜 주는 블록이다. 또한 버퍼 내에 명령어가 있는지 없는지를 항상 확인하여 신호를 발생하며, 분기 명령어 패취에 따른 분기 동작이 일어날 때 타겟 명령어가 버퍼 범위 안에 있는지 여부를 판단한다.

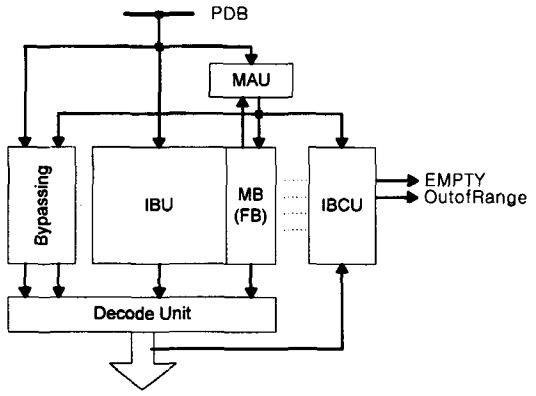


그림 1. 명령어 길이 정보를 포함하는 명령어 버퍼 블록도

Fig. 1. Block diagram of an instruction buffer including an instruction length information.

1. IBU

명령어 버퍼는 때론 여러 경우에 프로세서 성능을 현저히 떨어뜨릴 수 있다. 파이프라인 프로세서에서 연속적인 분기는 버퍼를 지우고 타겟 어드레스부터 새로 시작하도록 새로운 명령어를 패취해야 한다. 이는 파이프라인의 깊이(depth)에 따라 여러 사이클을 더 필요로 하게 된다. 일반적으로 RISC에서 분기 동작은 모든 명령어의 약 15%가량으로 시스템 성능에 중요한 영향을 미칠 수 있다. 즉, 각 루프에서 k사이클을 낭비하는, n개의 루프를 갖는 프로그램은 k*n 사이클을 낭비한다. 통계적으로 근거리 분기가 자주 일어나는 명령어 스트림을 수행할 때 16비트 명령어 길이를 기준으로 분기의 20-30%가 근거리 후방향(backward) 분기(48바이트 이하)이며, 10-20%가 근거리 전방향(forward) 분기(16바이트 이하)가 일어나는 것으로 알려져 있다[12]. 따라서 IBU에 저장되어 있는 명령어의 미스 율을 낮추기 위한 제어가 필요하며, IBU를 제어함으로써 프로세서가 직접 IBU 내에서 근거리 루프를 수행하도록 한다. 버퍼 크기는 전형적인 명령어 스트림의 통계적인 분석에 따라 후방향, 전방향 분기를 모두 고려하여 128바이트, 즉 32개의 명령어를 포함할 수 있도록 한다. 그러나 본 논문의 명령어는 16비트, 32비트 길이의 명령어를 가지고 있으므로 32~64개 명령어를 포함할 수 있다. 그림 2는 IBU 블록도이다.

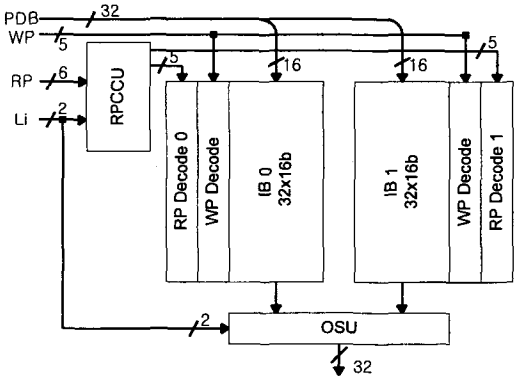


그림 2. IBU 블록도

Fig. 2. Block diagram of IBU.

IBU는 한 개의 WP(Write Pointer)와 두 개의 RP(Read Pointer)를 가지고 있다. 외부 메모리로부터 32비트 단위로 입력된 명령어를 16비트씩 각각 IB0, IB1에 저장한다. 따라서 WP는 IB0, IB1에 똑같이 적용되며, 총 6비트중 MSB 5비트만 IBU로 입력된다. WP는 항상 짝수로 증가하기 때문이다. 그러나 버퍼에서 명령어를 읽을 때는 명령어 길이에 따라 IB0, IB1의 포인터가 각각 달라져야 하므로 RP는 각각 따로 동작한다. RPCCU(RP Count Control Unit)은 IBCU에서 입력받은 6비트 RP 정보를 명령어 길이 정보에 따라 RP0, RP1을 출력한다. OSU(Output

Select Unit)는 IB0, IB1으로부터 출력된 16비트 길이의 명령어를 조합하여 디코더에 보낼 32비트 명령어를 만들어 낸다.

IB로 입력되는 가변 길이 명령어는 최초에 IB0의 짝수 어드레스에 저장된다. 물론 IB0에 저장된 16비트 명령어 다음에 32비트 명령어는 IB1의 홀수 어드레스에 저장될 수도 있다. 즉, RP 값이 '1' 증가하면 다음 명령어가 16비트 길이 명령어임을 가리키며, '2'가 증가하면 다음 수행될 명령어가 32비트 길이 명령어임을 나타낸다. 따라서 RPCCU는 16비트 길이 명령어와 32비트 명령어를 구분하여 각각 IB0, IB1의 RP값을 '1' 증가시키거나 정지(stall)시킨다.

IB를 제어하는데 핵심이 되는 것은 정확한 WP, RP를 발생하는 것이다. WP는 외부 메모리로부터 패취된 명령어가 저장될 IB상의 위치를 가리키며, RP는 IB로부터 출력될 명령어의 위치를 가리킨다. 또한 전방향, 후방향 분기 동작이 있을 때 타겟 명령어가 IB 안에 있는지 여부를 IBCU가 판단하여 타겟 명령어의 수행이 가능하게 한다. 타겟 명령어가 IB 안에 없다면 두 포인터는 '0' 값으로 리셋되고, 참조할 타겟 명령어를 외부 메모리에서 가져온다.

한 번 이상의 후방향 분기가 성공적으로 일어나면 RP는 WP보다 낮은 포인터 값을 유지하면서 IB에서 명령어를 출력하며, WP 위치에는 외부 메모리로부터 패취된 명령어가 계속 저장될 것이다. 따라서 또 한 번의 분기 동작이 일어나지 않는다면 WP는 IB내에서 최상위 포인터 위치까지 증가한 후 최하위 포인터 위치부터 다시 명령어를 쓰기 동작함으로써 이미 저장되어 있는 다른 명령어를 갱신시킬 것이다. 그러나 근거리 후방향 분기에 따른 IB내 명령어를 재사용할 수 있도록 최근에 사용한 명령어는 보존해야 한다. 따라서 WP는 RP와 일정 범위 내의 차이를 유지하도록 한다. 표 1은 명령어 길이 정보를 나타낸다.

표 1. 명령어 길이 정보 표현
Table 1. Expression of instruction length information.

Instruction Length IL(n)	32비트	16비트
	0	1
Mark Pointer MP(n)	Left(IBO)	Right(IB1)
	0	1

명령어 길이 정보는 명령어 길이와 명령어 시작점으로 구성된다. 명령어 길이는 32비트와 16비트 중 하나이며, 시작점은 IB0일 경우 '0'이며, IB1일 경우 '1'로 나타낸다. 예를 들어, 32L은 32비트 길이 명령어로 IB0, IB1에 각각 상위 16비트와 하위 16비트가 저장됨을 나타낸다. 16R은 16비트 명령어로 IB1에 저장됨을 의미한다. 그림 3은 OSU 구조를 나타낸다.

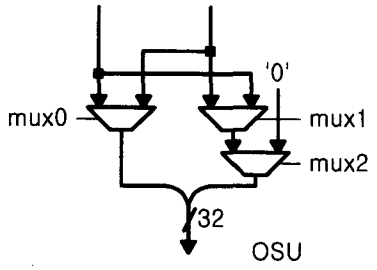


그림 3. OSU 블록도
Fig. 3. Block diagram of OSU.

OSU는 IB0, IB1에서 출력되는 값이 16비트 명령어인지, 32비트 명령어인가에 따라 32비트 결과를 만들어 내는 블록이다. 16비트 명령어나 32비트 명령어 모두 IB0, IB1에서 시작할 수 있으며, 위 그림의 다중화기는 IB0, IB1 출력을 적당한 순서로 재배열하여 32비트 명령어를 출력하는 역할을 한다. 표 2는 각각 16비트 명령어와 32비트 명령어의 출력 위치에 따라 다중화기 제어 신호를 나타낸 것이다. 16L, 16R은 각각 IB0, IB1에서 출력되는 값을 의미하며, 하위 16비트는 '0'으로 채워진다. 표 2는 32L, 16L, 32R, 16R 명령어를 출력하기 위한 그림 3의 OSU 다중화기 제어 신호 구성이다.

표 2. OSU의 다중화기 제어 신호
Table 2. Signals of multiplexer control in OSU.

내용	mux 0	mux 1	mux 2
32L	0	0	0
16L	0	0	1
32R	1	1	0
16R	1	1	1

mux0, mux1은 항상 같은 신호로 구동되며, mux2는 명령어의 길이를 구분하여 16비트 명령어일 경우 16L이나 16R 값을 상위 16비트로 하고, 하위 16비트는 '0'으로 채우게 된다. 32비트 명령어일 경우 mux0,

mux1은 32L, 32R에 따라 OSU로 입력된 두 16비트 값을 조합하는 역할을 한다.

2. IBCU

그림 4는 IBCU 블록도이다.

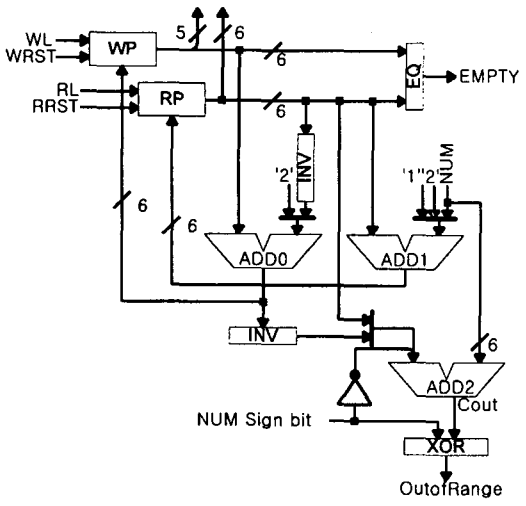


그림 4. IBCU 블록도
Fig. 4. Block diagram of IBCU.

RP, WP 값의 증감은 덧셈기를 사용한다. WP는 매 사이클마다 증가하여 새로운 명령어를 IB에 저장한다. RP는 IB로부터 명령어를 읽기 위한 포인터이다. WL과 RL은 두 포인터에 새로운 값을 로드하기 위한 제어 신호이며, WRST, RRST는 각 포인터를 리셋하기 위한 신호이다. 이와 같은 리셋신호는 분기 동작의 타겟 명령어가 IB 내에 존재하지 않을 때 외부에서 명령어를 다시 읽어올 경우 발생한다. 결국 IB의 최하위 위치부터 다시 명령어 읽기, 쓰기 동작을 시작하는 것이다.

덧셈기는 RP, WP를 증감시키기 위한 것 뿐만 아니라 WP와 RP사이의 차이 값을 연산한다. 한 번 이상의 분기 동작에 의해 WP, RP는 서로 다른 위치를 가리키게 된다. 따라서 두 포인터의 차이는 WP와 RP 사이에 존재하고 있는 명령어의 개수를 가리키며, 그 명령어들은 IB내에 있으면서 아직 수행되지 않은 명령어이다. 그러나 이 차이 값은 실제 명령어 개수보다 '1'이 작다. WP는 다음 사이클에 저장할 명령어의 IB 내 위치를 가리키며, 따라서 다음 명령어는 아직 저장되어 있지 않기 때문이다. 이와 같은 연산은 1의 보수 덧셈을 이용한다. 즉 RP를 인버팅하여 WP와 더하는 것이다. ADD0의 출력을 다시 인버팅한 결과는 아직 갱신되지 않았거나 IB내 채워지지 않은 명령어 수이다.

분기가 IB 범위를 벗어날 때 WP, RP는 '0'으로 리셋되고, 명령어는 외부 메모리에서 다시 패취된다. 만약 순차적인 명령어 수행 중 NUM 만큼의 분기 동작이 일어나면 ADD2 입력으로 RP가 선택될 것이다. 왜냐하면 RP로부터 오프셋만큼 분기했을 때 ADD2의 캐리 값을 이용하여 타겟 명령어의 유무를 확인할 수 있기 때문이다.

NUM 값은 6비트 오프셋 값으로 -16~15의 분기 영역을 가리킨다. 왜냐하면 NUM 값은 외부 프로그램 메모리의 오프셋이 아니라 16비트 단위의 IB0, IB1을 모두 가리키는 포인터이기 때문이다. 따라서 프로그램 메모리의 새로운 PC값을 연산할 때는 $\frac{NUM}{2}$ 으로 연산해야 한다. -16~15 이상에서 분기가 일어날 때는 32비트 명령어에서 16비트 길이의 오프셋을 사용한다. 따라서 직접 모드(immediate mode)로 접근할 수 있는 메모리는 -2^{21}~-2^4-1, $+2^4$~2^{20}-1 범위를 갖는다. 즉 16비트에 오프셋 5비트가 더해진 범위를 갖는다.

그러나 최근에 분기 동작이 일어난 상황에서 다시 분기가 일어날 때 IB 범위를 넘는지, 않는지를 판단할 때는 RP 값 대신에 ADD0 결과를 인버팅한 값을 사용한다. 인버팅한 결과 값은 WP, RP 차이 이외의 범위를 가리키며, ADD2의 입력이 될 경우 앞서 발생한 분기 범위 이외의 영역에서 다시 한 번 오프셋만큼의 분기를 했을 때 타겟 명령어의 IB 범위 이탈 여부를 알 수 있기 때문이다. ADD1은 새로운 RP값을 연산한다. 분기가 일어나지 않을 때는 단순히 '1' 혹은 '2'씩 증가하지만 분기가 일어났을 때는 IB내에 타겟 명령어가 있으면 타겟 위치를 발생시킨다. 이때 ADD1은 RP값과 6비트의 2의 보수 값을 갖는 오프셋 값이 입력된다. ADD2는 분기 명령어가 IB내에 있는지 없는지를 판단하기 위해 사용한다. ADD2의 Cout 신호는 IB 범위내에 분기 명령어가 있는지 없는지를 판단해 준다. 표 3은 Cout, NUM값에 따른 명령어의 IB 범위 이탈 여부를 보여주고 있다.

NUM이 양수일 때는 ADD2의 또 다른 입력 부호를 '1'로 강제하며, NUM이 음수일 때는 또 다른 입력 부호를 건드리지 않는다. Cout은 NUM의 부호 비트와 XOR하며, 타겟 명령어가 IB 범위 밖에 있으면

OutofRange신호는 '1'이 된다. 그림 5는 IB 제어 기능을 보여주고 있다.

표 3. Cout, NUM 신호에 따른 IB 범위 이탈 여부 판정

Table 3. Indication of the range is given by the carry-out signal Cout with offset signal NUM.

Cout	NUM > 0	NUM < 0
0	In the range	Out of the range
1	Out of the range	In the range

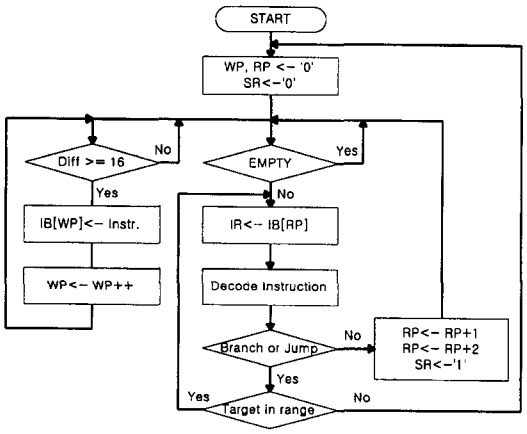


그림 5. 명령어 버퍼 제어 기능 흐름도

Fig. 5. Flow chart illustrating the function of the instruction buffer control.

WP, RP, SR(Status Register)은 '0'으로 초기화된 후 IB가 비어 있는지 여부를 판단한다. SR은 분기 여부가 일어났는지 여부를 나타내는 레지스터이다. 따라서 분기가 일어나면 이 값은 '1'이 된다. 초기에 IB는 레지스터가 초기화되기 때문에 비어 있다. 따라서 분기 명령어를 패취하지 않는 한 IB가 모두 채워질 때까지 WP는 '2'씩 증가한다. 이때 RP도 함께 증가하면서 명령어를 디코더로 출력한다. 그리고 나서 패취된 명령어의 분기 여부를 판단하여 분기가 일어나면 타겟 명령어가 버퍼안에 있는지 여부를 판단한다. 버퍼 안에 타겟 명령어가 존재하지 않으면 WP, RP, SR을 다시 초기화하며, 버퍼 안에 타겟 명령어가 존재하면 갱신된 RP가 가리키는 곳의 타겟 명령어를 출력한다. WP는 항상 RP 값을 참조하여 패취동작을 한다. 만약 RP와 '16'이상의 차이가 날 때만 패취 동작을 하고, 그렇지 않으면 패취 동작을 멈추도록 한다. '16' 이내의 차이는 근거리 전방향 분기 동작이 일어날 때 타겟 명령어가 존재할 범위이다. 따라서 새로 패취된 명령어에 의해 타겟 명령어가 될 수 있는 명령어가 갱신되어 없어지는 것을 막기 위한 것이다[12].

3. MAU

MAU는 전 상태의 명령어 길이 정보 MB [n-1] 와 명령어 그룹 필드 GF [n] 에 따라 결정된다. 그림 6은 명령어 길이 정보를 만드는 블록도이다.

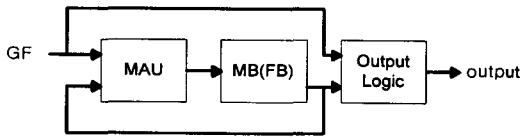


그림 6. 명령어 길이 정보 출력기 블록도

Fig. 6. Block diagram of output of instruction length information.

MB(Mark Buffer)는 MAU에서 만들어진 명령어 길이 정보를 저장하는 버퍼로 64×2b 크기를 가지며, 길이 정보 유효 여부를 판단해 주는 64×1b 크기의 FB(Flag Buffer)를 포함한다. 현재 명령어 길이 정보를 구할 때 전 상태의 MB 유효 여부가 필요하므로 MB값을 구한 후 유효 여부를 판단하여 FB에 저장한다. 그림 7은 명령어 길이 정보의 변화 및 상태도를 나타낸 것이다.

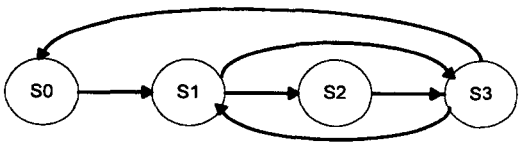


그림 7. 명령어 길이 정보 상태도

Fig. 7. State diagram of instruction length information.

S_0는 32L 상태, S_1는 16L 상태, S_2는 16R 상태, S_3는 32R 상태를 나타낸다. S_0는 32L 상태로 32L 상태나 S_1인 16L 상태로 변할 수 있다. S_1은 16L 상태로 다음 명령어가 32비트 길이이면 32R 상태로, 16비트 길이 명령어이면 16R 상태로 변한다. 그림 8은 그림 7의 상태 변화에 따라 이전 명령어의 길이 정보를 참조하여 현재 명령어 길이 정보를 구하는 블록도이다.

L_{iL}, L_{iR}은 IB0, IB1에 각각 저장되는 명령어 길이 정보이다. Li는 2비트 정보로 상위 비트는 명령어 길이를 나타내는 L, 하위 비트는 명령어 시작점을 나타

내는 MP(Mark Pointer)로 구성된다. F_A, F_B, F_C, F_D는 명령어 길이 정보의 유효 여부를 나타내는 상태 플래그이다. 16비트 명령어 그룹 필드는 "10"으로 정의하며, IL과 MP는 (1)과 같이 구할 수 있다.

$$L(n) = GF(1) \text{ AND } (\text{ NOT } GF(0)) \quad (1)$$

$$MP(n) = MP(n-1) \text{ XOR } L(n-1)$$

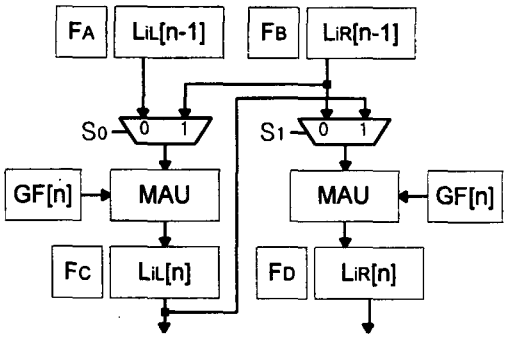


그림 8. 현재 명령어 길이 정보 출력 블록도
Fig. 8. Block diagram of present instruction length information.

따라서 Li(n)는 현재 명령어의 명령어 시작점 위치인 MP(n)값을 구한 후 32비트 입력 필드 중 [31:30]과 [15:14] 중 하나를 선택하여 그룹 필드 정보 GF [1:0] 로 사용하여 L(n) 값을 구한다. 또한, 그림 8의 각 플래그 및 이전 명령어의 길이 정보를 선택하는 신호 S_0, S_1은 (2)와 같이 구할 수 있다.

$$F_C = (F_A \text{ AND } (\text{NOT } L_A)) \text{ OR } (F_B \text{ AND } L_B)$$

$$S_0 = \text{NOT } (F_A \text{ AND } (\text{NOT } L_A)) \text{ OR } (F_B \text{ AND } L_B) \quad (2)$$

$$F_D = (F_B \text{ AND } (\text{NOT } L_B)) \text{ OR } (F_C \text{ AND } L_C)$$

$$S_1 = \text{NOT } (F_B \text{ AND } (\text{NOT } L_B)) \text{ OR } (F_C \text{ AND } L_C)$$

F_C는 이전 명령어의 길이와 유효 여부에 의해 결정된다. 예를 들어, 스택 명령어가 아니고(L_A=0), F_A가 유효하거나 스택 명령어 이면서 F_B가 유효하다면 다음 명령어는 IB0의 출력이 상위 16비트가 되는 32비트 길이의 명령어가 되므로, F_C는 '1'이 되며, 그 이외의 값들은 '0'이 된다. F_D, S_1은 F_C, L_C 결과가 나온 후 결정된다.

이와 같이 명령어 길이 정보를 구한 후 그 정보에 따라 IB에서 명령어를 패취하기 위한 RP값의 증감이 필요하다. 표 4는 위와 같은 상태 변화에 따른 RP값의 변화 상태를 나타낸 것이다.

표 4. 상태 변화에 따른 RP 값의 변화
Table 4. RP variation according to state variation.

상태변화	RP0	RP0 버스	RP1	RP1 버스	어드레스 증가
32L→32L	+	RP[5:1]	+	RP[5:1]	+2
32L→16L	+	RP[5:1]	.	RP[5:1]	+1
32R→32R	+	RP[5:1] +1	+	RP[5:1]	+2
32R→16R	.	RP[5:1]	+	RP[5:1]	+1
16R→32L	+	RP[5:1]	+	RP[5:1]	+2
16R→16L	+	RP[5:1]	.	RP[5:1]	+1
16L→32R	+	RP[5:1] +1	+	RP[5:1]	+2
16L→16R	.	RP[5:1]	+	RP[5:1]	+1

예를 들면 명령어 길이 정보 변화가 32L→32L인 경우는 전 명령어가 32비트 명령어로 IB0 16비트가 상위 16비트이고, 현재 명령어가 32비트이면 RP 0, 1 모두 '1'씩 증가한다. 32R→16R은 전 명령어가 32비트 길이 명령어로 IB1의 16비트가 상위 16비트이며, 현재 명령어는 16비트 길이 명령어로 IB1에서 출력되어야 한다. 따라서 RP0는 증가하지 않고, RP1만 '1' 증가한다. RP0, RP1 버스는 32R인 경우를 제외하고는 모두 6비트 RP중 [5:1] 비트가 RP 값이 된다. 하지만 32R은 왼쪽 IB0에서 RP의 한 포인트가 증가한 값을 출력하기 때문에 RP [5:1] 값에 '1'이 증가된 RP를 사용한다.

4. PC 연산

RP는 프로세서 내에서는 16비트 명령어 길이 단위로 증감한다. 따라서 IB에 저장되어 있는 명령어를 읽기 위한 RP 값은 16비트 단위로 연산된 결과이다. 그러나 외부 프로그램 메모리의 어드레스일 경우는 두 가지로 동작을 한다. 첫 번째, RP 값이 홀수일 때는 LSB 1비트를 제외한 어드레스에 '1'을 더한 어드레스에서 명령어를 패취한다. 두 번째, RP 값이 짝수일 때는 전 명령어가 32비트일 때는 LSB 1비트를 제외한 어드레스에서 패취를 하며, 전 명령어가 16비트일 때는 패취 동작을 정지시킨다. 이는 이미 16비트 전 명령어에 의해 현재 PC에 해당하는 명령어가 이미 패취된 상태이기 때문이다. 표 5는 RP값의 변화에 따른

프로그램 메모리 어드레스 변화의 예를 보여주고 있다.

표 5. RP변화에 따른 프로그램 메모리 어드레스 변화

Table 5. Program memory address variation according to RP variation.

프로그램 메모리 어드레스	RP 위치	
0	0	
1	2	3
2		5
3		7
4	8	
5	10	

RP 값이 프로세서 내에서 0→2→3→5→7→8→10으로 변할 때 외부 프로그램 메모리 어드레스는 0→1→2→3→4→'stall'→5→ 순서로 바뀐다.

Ⅲ. VHDL 설계 및 합성

본 논문의 명령어 버퍼 구조는 각 모듈 블록을 VHDL[13][14][15]을 이용하여 설계한 후 COMPASS ASIC Synthesizer 툴을 이용하여 합성하였다. 그림 9는 VHDL을 이용한 ASIC 설계 흐름도를 보여주고 있다.

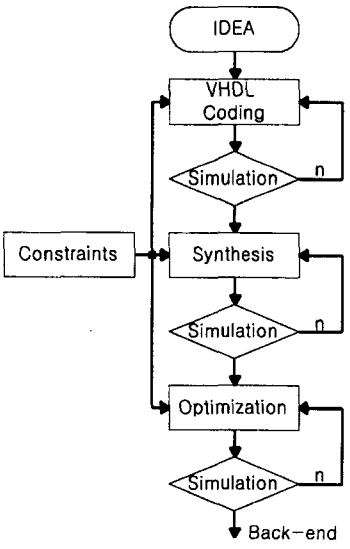


그림 9. VHDL을 이용한 ASIC 설계 흐름도

Fig. 9. Flow chart of ASIC design using VHDL.

VHDL 코드의 구조적 수준 및 행위적 수준에서 기술 및 시뮬레이션은 각 모듈을 Model Technology Inc.의 V-System PC 버전 4.4를 이용하였다. 사용한 라이브러리는 V-System에서 기본으로 제공하는 IEEE 라이브러리만을 사용하였다. 논리 합성은 VHDL을 최적화된 게이트 수준의 네트리스트로 변환하는 0.6㎛ COMPASS ASIC Synthesizer 툴을 이용하였다. 시뮬레이션을 위한 VHDL 코드는 3개의 컴포넌트, ibu, ibcu, mau로 구성되어 있다. ibu는 IB, MB, FB등 버퍼를 포함하는 컴포넌트이며, ibcu는 RP, WP 값을 발생하고, IB 범위내에 명령어가 있는지 여부를 판단하는 컴포넌트이며, mau는 입력된 명령어의 길이 정보를 추출하기 위한 컴포넌트이다. 합성 과정에서는 면적의 최적화를 이루기 위한 네트리스트를 생성하도록 하며, 동작 속도는 최적화 과정을 반복하여 합성 과정이 이루어지도록 하는 제한 조건(constraints)을 두었다. ASIC Synthesizer 툴을 이용해 합성한 결과는 게이트 수 17,000여개와 7ns의 최장경로를 형성한다[16]. 최장 경로는 그림 4의 ADD0, ADD2를 통해 연산이 연속적으로 이루어질 때 발생한다.

Ⅳ. 시뮬레이션 결과 및 검토

그림 8은 (3)과 같은 16비트, 32비트 명령어가 섞여 있는 프로그램을 시뮬레이션한 결과이다.

```
'h02000000      ADDC
'h81f0          SWAP
'h12000000      MOV          (3)
'h14000000      AND
'h91f0          DUP
'h12000000      MOV
'h40000000      SLADD
```

L_{iL}, L_{iR} 신호는 패취된 명령어의 길이 정보를 나타내며, f는 버퍼 내 명령어의 유효 여부를 나타내는 플래그 신호이다. pdb는 외부에서 패취되어 입력되는 프로그램 데이터 신호이고, out 신호는 IB 출력 값이다. 또한 empty, outofrange 신호는 ibcu 블록에서 만들어지는 신호이다.

외부 프로그램 메모리에서 내부 IB로 입력되는 pdb 데이터는 (4)와 같은 순서로 입력된다.

'h02000000 → 'h81f01200 → 'h00001400 → 'h000091f0

→ 'h12000000 → 'stall' → 'h40000000 (4)

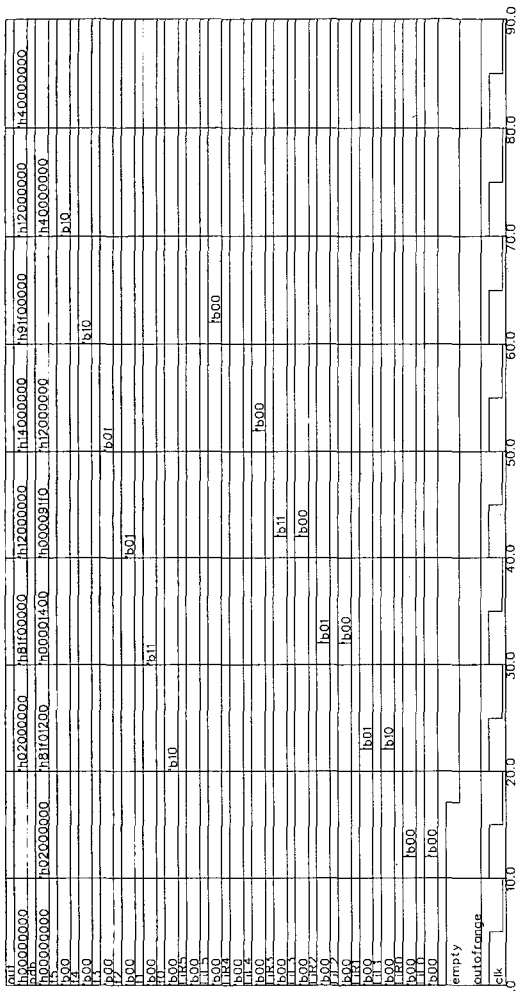


그림 10. 명령어 버퍼의 시뮬레이션 결과
Fig. 10. Test simulation result of an instruction buffer.

L_{iL}, L_{iR} 출력 값은 명령어 길이 정보로 명령어 패취와 동시에 그 값이 출력된다. 예를 들어 L_{iL1}, L_{iR1}은 10ns 이후 각각 '10', '01'값을 갖는다. 이는 외부 메모리로부터 패취된 'h81f01200'에 대한 명령어 길이 정보이다. 상위 16비트는 'SWAP' 명령어로 왼쪽 버퍼로부터 출력되며, 하위 16비트는 32비트 명령어의 상위 16비트가 됨을 의미한다. f값은 레지스터 출력이기 때문에 전 사이클에 패취된 명령어의 유효 여부를 판정하는 신호이다. 예를 들어 f1값은 '11'로 이는 'h81f01200' 데이터에 대한 판정이라고 할 수 있다. 즉, 상위 16비트와 하위 16비트 모두 명령어 시작 부분임을 의미한다. 출력 데이터는 (3)의 순서대로 매 사이클마다 출력되며, 16비트 명령어의 경우는 하위 16비트가 '0'으로 채워져 32비트가 되어 출력된다.

표 6은 본 논문의 명령어 버퍼와 여러 프로세서 내 캐쉬 및 버퍼 크기 등을 비교한 것이다.

표 6. 명령어 버퍼의 성능 비교
Table 6. Comparison of Instruction Buffer Dimensions.

	버퍼 크기(byte)	명령어 형태	명령어 캐쉬 크기(KB)	성능(MHz)
A[5]	8	고정	NA	50
B[6]	16	고정	32	233
C[7]	92	고정	32	160
D[17]	12	가변	4	25
E[18]	32	고정	8	40
Ours	128	가변	NA	140

표 6의 모든 프로세서는 내부나 외부에 캐쉬를 가지고 있으며, 명령어 버퍼는 명령어를 디코더로 보내기 위한 FIFO(First In First Out)나 우선패취 용도로만 사용하기 때문에 대체적으로 본 논문의 버퍼 크기보다 더 작다. 또한 명령어 길이가 고정되어 있기 때문에 명령어의 길이를 구분할 필요가 없다. 또한 복잡한 구조를 갖는 캐쉬의 경우 2사이클 이상의 파이프라인으로 구성되는 반면, 본 논문의 명령어 버퍼는 단일 사이클에 동작이 가능하다. 따라서 최고 140MHz까지의 성능을 갖는 프로세서에 내장될 수 있으며, 내장된 캐쉬가 없더라도 -16~15범위의 오프셋을 갖는 분기 동작에 대해서는 타겟 명령어를 찾을 수 있는 구조이다. 따라서 본 명령어 버퍼는 칩의 상당 부분을 차지하는 내부 캐쉬를 내장하지 않고도 근거리 분기 동작에 대처할 수 있는 성능을 가지고 있으며, 16비트, 32비트의 가변 길이 명령어를 동시에 처리할 수 있기 때문에 스택과 레지스터에 동시에 기반하는 프로세서 응용에 적당한 구조라고 할 수 있다.

V. 결 론

본 논문에서는 네트워크 응용의 발전에 따라 빠른 서브루틴 호출 등의 기능이 가능하도록 스택을 이용한 객체 지향형 데이터 처리를 하는 16비트 명령어와 레

지스터를 사용한 벡터 데이터를 지원하는 32비트 명령어를 동시에 저장하고, 처리할 수 있는 명령어 버퍼를 구현하였다. 본 구조는 명령어 버퍼에 저장되어 있는 가변 길이 명령어의 미스율을 낮출 수 있도록 외부 메모리에서 패취되었을 때 명령어의 길이 정보를 동시에 발생시킨다. 분기 동작에 따른 히트율을 높이기 위해 통계적으로 산출된 전방향, 후방향 분기가 가능한 버퍼 크기를 가지며, 외부 메모리로부터 계속적인 패취로 인한 최근 수행된 명령어나 아직 수행이 되지 않은 명령어의 갱신을 방지하기 위해 읽기, 쓰기 포인터 차이를 일정 범위 이내에서 유지하도록 제어한다. 또한 근거리 루프 동작은 버퍼 내에서 직접 수행이 가능하며, 타겟 명령어의 히트 및 미스 여부를 판단하는 회로를 내장하고 있다. 외부 메모리에서 패취된 모든 명령어는 각각 길이와 버퍼 내 시작점을 알려 주는 정보가 만들어지며, 명령어 디코더는 이 정보를 참조하여 순차적인 명령어 동작이 가능하도록 한다.

가변 길이 명령어를 처리하기 위한 각 모듈은 Model Technology Inc.의 V-System를 사용하여 설계 및 시뮬레이션하였으며, 합성 및 검증은 COMPASS ASIC Synthesizer 툴을 사용하였다. 최장경로는 약 7ns로 최고 동작 속도는 약 140MHz까지 가능하다. 총 게이트 수는 약 17,000개이다.

본 논문의 결과는 네트워크 환경에서 발생할 수 있는 객체 지향형 영상 데이터 처리를 위한 프로세서의 명령어 버퍼로 이용될 것이며, 16비트 명령어 만으로 이루어진 프로세서에서도 응용이 가능하다.

참 고 문 헌

[1] T. Kato, et. al., "Instruction method and execution system for instructions including plural instruction codes", United States Patent 5,442,762, Feb. 1994.

[2] M. Esponda and R. Rojas, "The RISC Concept - A Survey of Implementations", http://www.inf.fu-berlin.de/lehre/WS94/RA/RISC-9.html, Sept. 1991.

[3] P. J. Koopman, Jr., "Stack Computers: the new wave", http://www.cs.cmu.edu/~koopman/stack_computers/, 1989.

[4] W. Stallings, *Computer organization and Architecture : Principles of Structure and Function*, 3rd, Macmillan Publishing Company, pp. 391-397, 1993.

[5] R. Niles, "Motorola MC68060 Information", http://www.omnipresence.com/Amiga/News/AR/AR214_Sections/P1-8.HTML, Apr. 1994.

[6] "AMD K6", http://www.fastgraphics.com/k6.html.

[7] P. Song, "IBM's Power3 to Replace P2SC, Microprocessor Report Volume 11, Number 15", http://www.chipanalyst.com/q/@2585477cchctf/report/samples/1115/111507.html, Nov. 1997.

[8] Sun Microsystems, Inc., "picojava™ I Microprocessor Core Architecture", http://sunsite.ics.forth.gr/sunsite/mirror1/sun_microelectronics/picojava/wpr-0014-01/, Nov. 1996.

[9] D. J. Shippy, T. W. Griffith, and G. Braceras, "POWER2 Fixed-Point, Data Cache, and Storage Control Units", http://www.fudan.sh.cn/hkuweb/training/html/ibmhwsw/fxu.html#link_3, 1994.

[10] Motorola, Inc., "MOTOROLA AIMS ColdFire(tm) MCF5204 AT NEW EMBEDDED MARKETS", http://www.mot-sps.com/press/html/PR960812B.html, Aug. 1996.

[11] A. Yoshitake, et. al, "Data processor decoding and executing a train of instructions of variable length at increased speed", United States Patent 5,581, 774, Mar. 1994.

[12] V. G. Oklobdzija, et. al, "Instruction prefetch buffer control", United States Patent 4, 714, 994, Dec. 1987.

[13] S. Mazor and P. Langstraat, *A Guide to VHDL*, Kluwer Academic Publishers, 1993.

[14] V-System/VHDL PC User's Manual Version 4.4, Model Technology, 1996.

[15] D. L. Perry, *VHDL*, McGraw-Hill, 1991.

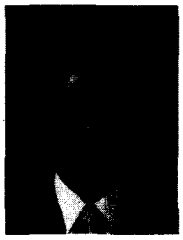
[16] *COMPASS Tool에서의 VHDL을 이용한 ASIC 설계*, 반도체설계교육센터, 1996년 1월

[17] J. Circello, "ColdFire: A Hot Architecture", http://www.byte.com/art/9505/

sec13/art1.htm#coldfire, May 1995.

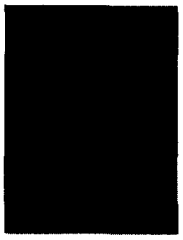
[18] Y.Saito, et. al, "A 1.71M-Transistor CMOS CPU Chip with a Testable Cache Architecture", ISSCC93 Digest of Technical Papers, pp. 86-87, Feb. 1993.

저 자 소 개



朴 柱 炫(正會員)

1969년 7월 13일생. 1993년 2월 전남대학교 전자공학과 졸업. 1995년 2월 전남대학교 대학원 전자공학과 졸업(공학석사). 1997년 2월 전남대학교 대학원 전자공학과 수료(공학박사). 1998년 2월 ~ 현재 전남대학교 공과대학 전자통신기술연구소(ETTRC) 전임연구원. 홈페이지 주소 : http://chonnam.chonnam.ac.kr/~pjh. 주관심분야는 MPEG 코덱 설계, 스택 머신, 객체 지향 프로세서, DSP, RISC 프로세서 설계 등임



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). 1986년 오하이오 주립대학교 전기공학과(공학박사). 1988년 6월 ~ 1991년 8월 한국전자통신연구소(실장). 1991년 9월 ~ 현재 전남대학교 전자공학과 교수. 1998년 2월 ~ 현재 전남대학교 공과대학 전자통신기술연구소(ETTRC) 소장. 1997년 12월 ~ 현재 반도체설계교육센터(IDEC) 전남대학교 지역센터장. 주관심분야는 ADSL 모뎀 설계, MPEG2, MPEG4 시스템 설계 등임