5 단계 파이프라인 DSP 코어를 위한 시뮬레이터의 설계

김문경, 정우경, 문상국, 안상준, 이용석
연세대학교 전자공학과 VLSI & CAD 연구실
서울시 서대문구 신촌동 134 번지
bungae@bubble.yonsei.ac.kr

A Simulator for a Five-stage Pipeline DSP core

Moongyung Kim, Wookyeong Jeong, Sangook Moon, Sangjun An, Yongsurk Lee
Elec. Eng., Yonsei University
134, Shinchon-dong, Seodaemun-gu, Seoul
bungae@bubble.yonsei.ac.kr

Abstract

We designed a DSP core simulator with C language, that is able to simulate 5-stage pipelined DSP core, named YS-DSP. It can emulate all 5 stage pipelines in the DSP core. It can also emulate memory access, exception processing, and DSP parallel processing. Each pipeline stage is implemented by combination of one or more functions to process parts of each stage. After modeling and validating the simulator, we can use it to verify and to complement the DSP core HDL model and to enhance its performance.

1. 서 론

처리 정보의 양이 늘어나고 있는 현재에는 좀더 복잡한 처리가 가능하고 고속의 처리능력을 가진 DSP 마이크로프로세서의 코어의 설계가 필요하다. 이러한 마이크로프로세서의 모델을 만들고 설계하기 위해서 일반적으로는 HDL(Hardware Description Language)를 이용한다. 이로 인해서 실제 마이크로프로세서를 만드는데 걸리는 시간이 많이 단축되었지만 원하는 성능을 거두기 위해서는 처음 설계한 모델에 많은 보완과 수정이 필요하게 된다.

그런데, 이러한 모델을 작성하고 수행하는데 걸리는 시간은 예전의 하위구조부터 전체를 직접 설계하는 방식보다는 빠르나 추가 및 변경과 그로 인한 성능향상이 어느 정도인지 알아보는 데는 적지 않은 시간이 걸리게 된다. 따라서 이러한 테스트를 수월케 하기 위해서 C 언어를 이용하여 목적한 구조의 DSP 코어의 모델을 만드는 방식을 채택하게 되었다.

여기서는 이를 기본으로 한 시뮬레이터를 설계, 목적한 구조의 DSP 코어를 모델링 하여 원래 목적한 동작을 하는지 살펴보아 그 모델링이 정확했는지 살펴보고 실제 설계 중인 HDL 모델과의 동작비교를 통해 두 모델의 동작이 동일한지 확인한다. 또한 이와 같은 방식으로 검증된 두 모델의 동일성을 이용하여 C 언어를 사용한 모델에서 성능의 향상을 가져올 수 있는 부분들, 예를 들면 버퍼의 크기나 기능 블록의 통합 및 추가, 삭제 등을 통해 실제의 성능향상 정도를 알아보고 여기서 향상을 야기시킨 구조의 변경은 HDL 모델에 적용하여 목표 구조를 향상시킬 수 있게 한다.

2. YS-DSP 코어의 사양

이 시뮬레이터의 모델이 되는 YS-DSP의 설계 사양은 다음과 같다. 이는 32 비트 마이크로 프로세서 구조를 가지며 크게 RISC 유닛과 DSP 유닛으로 이루어져 있다. 모든 제어는 RISC 유닛을 중심으로 이루어지고 DSP 유닛은 디지털 신호 관련 처리를 중점적으로 하도록 설계되었다.

파이프라인 구조는 FETCH - DECODE - EXECUTE - MEMORY – WB_DSP의 기본적인 5단 파이프라인 구조를 가진다. YS-DSP는 RISC 유닛과 DSP 유닛이 각각의 레지스터 파일을 가지고 있기 때문에 일반적인 RISC 유닛의 명령의 경우에 메모리를 읽고 쓸 필요가 없을 경우에는 EXECUTE 단에서 종료가 된다. 그리고 DSP 유닛의 경우에는 WB_DSP 단에서 종료가 된다. 또한 여러 사이클을 필요로 하는 명령어의 경우에는 EXECUTE 단과 MEMORY 단을 더 필요로 하게 되므로 최고 7단계까지를 필요로 하는 경우도 있다. 이와는 별개로 곱셈기를 위한 MULTIPLY 단계가 존재한다. 이는 MEMORY 단에서 사용 중인지를 검사하고 사용중이 아닐 경우 WB_DSP 단 없이 곧장 MULTIPLY 단으로 넘어가 곱셈을 수행한다.

기본적으로 수행되는 RISC 명령어 이외에, DSP의 처리를 위한 DSP 명령어가 있는데 이는 고정 소수점 연산을 기본으로 한다. RISC 명령어는 16 비트 길이를

* 본 연구는 1998년도 한국 학술진흥재단 대학부설 연구소 연구비에 의하여 연구 되었음

가지고 DSP 명령어는 32 비트 길이를 가지는데, 그 중 하위 16 비트는 DSP 명령어를 가리키고 상위 16 비트는 MOVX, MOVY 의 데이터 이동 명령어를 가리키게 됨으로써 DSP 동작과 데이터 이동을 동시에 처리될 수 있기 때문에 고속을 요구하는 DSP 응용에 적합하다.

YS-DSP 코어의 내부에는 24KB 의 ROM 두 개와 4KB 의 RAM 두 개가 있는데, ROM 과 RAM 하나씩을 묶어서 X 메모리와 Y 메모리로 관리해 준다. DSP 유닛에서는 MOVX 와 MOVY 라는 명령어를 함께 수행시킬 수 있는데 이 경우에는 X 메모리와 Y 메모리로 나뉘어진 영역을 동시에 제어, 데이터를 읽어올 수 있다. 여기서, 전체 메모리 공간의 데이터 전송을 위한 32 비트 길이의 데이터 버스와 주소 버스가 존재하고, X 메모리와 Y 메모리 각각을 위한 16 비트 길이의 데이터 버스와 주소버스가 존재하며, 이로 인해서 동시에 데이터를 읽어오는 경우에는 한 워드길이(16 비트)만큼씩 만 읽어올 수 있다.

3. 시뮬레이터의 구성

본 연구에서 설계하는 시뮬레이터는 기본적으로 5 단계 파이프라인의 각 단계를 구현해 준 부분과 이를 제어하는 부분, 예외 처리 부분, 외부 인터페이스 부분으로 나뉘어 진다. 특히, EXECUTE 단과 WB_DSP 단은 처리해 줄 명령어들을 모아 둔 RISC 명령어 모듈과 DSP 명령어 모듈이 따로 존재하여 이의 호출을 중심 동작으로 하도록 구현되어 있다. 파이프라인을 제어하는 부분이 이 시뮬레이터의 중심부라고 할 수 있는데, 이는 각 단계를 호출하는 부분과 각 단계의 수행 중 생성된 제어신호를 처리해 주는 부분으로 구성되어 있다. 그리고 외부 인터페이스 부분은 사용자의 편의를 위해 결과를 표시해 주는 부분과 사용자의 제어 입력을 받아들이는 부분으로 구성되어 있다. 이의 간략한 형태는 다음에 나오는 그림 1.과 같다.

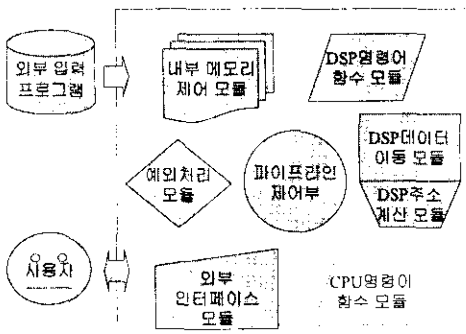

그림 1. 시뮬레이터의 개략적인 구조

동작 순서는 그림 2.에 나온 순서와 같다. 일단 매 사이클마다 PREFETCH 단에서 명령어를 미리 읽어서 PREFETCH 버퍼에 저장해 둔다. 그리고 FETCH 단에서 PREFETCH 버퍼에 있는 명령어 데이터를 읽어오게 되고 DECODE 단에서 이 명령어를 인식하여 인식된 명령어가 필요로 하는 인자들을 배정해 주고 EXECUTE 단에서 이 명령어를 수행해 준다. 만일 이 명령어가 메모리 제어를 필요로 한다면 MEMORY 단으로 가서 메모리의 데이터를 제어하고 DSP 동작이나 메모리에서 레지스터로의 기록을 요구한다면 WB_DSP 단에서 해당 DSP 명령어를 수행하던가 레지스터로의 기록을 수행한다. 그리고 여기서 곱셈 명령이 입력되었을 경우에는 MEMORY 단에서 곱셈기에 데이터를 보낸 후 다음 단으로 WB_DSP 단이 아닌 MULTIPLY 단으로 이동하여 곱셈동작을 지정된 사이클만큼 수행해 주게 된다.

3.1 파이프라인 각 단계의 구현

이 시뮬레이터에서 지원하는 파이프라인의 단계는 PREFETCH, FETCH, DECODE, EXECUTE, MEMORY, WB_DSP, MULTIPLY 의 7 단계로 되어 있고 실제 기본 동작은 PREFETCH 와 MULTIPLY 를 제외한 5 단계로 이루어지게 된다. 이들은 기본적으로 명령어의 수행을 목적으로 하기 때문에 시뮬레이터의 구현을 위해서는 각각의 명령어의 구현을 우선시 하게 된다. 각각의 명령어의 구현은 그 명령어의 동작만을 구현하면 되고 이때는 각 단계의 동작을 그리 고려하지 않아도 된다.

이렇게 구현된 명령어들을 구분 가능하도록 구분 단계를 만들어 주어야 하는데, 이 부분은 읽어 들인 명령어를 비트 단위로 분석, 어떤 명령어의 입력이 있는지를 알아내는 부분이 필요한데, 이를 위해 C 언어에 있는 비트 할당 기능과 데이터 공유기능을 이용해 준다. 그런데 이는 시뮬레이터의 구현이 어떤 자원을 가지고 이루어지는지를 알아야 한다. 일례로 본 논문에서 구현된 시뮬레이터는 Turbo C++ 3.0 을 이용, DOS 환경에서 구현되었다. UNIX 상에서 gcc 를 이용해 구현할 경우, UNIX 에서의 데이터들은 Big-Endian 의 적용을 받기 때문에 비트 할당을 할 때 데이터의 상위 비트부터 차근차근 할당을 하게된다. 그런데 DOS 상에서 구현할 때는 앞서의 경우와 반대가 되어, Little-Endian 의 적용을 받아서 데이터의 하위 비트부터 할당을 하도록 설계해 주어야 한다. 구현 시에 경험한 바로는 한 두 비트를 할당 할 경우에도 하위 비트가 먼저, 상위 비트가 나중이 되어야 했다.

이렇게 구현된 구분 단계에 앞서 구현한 명령어들을 호출 가능하도록 구성해 주면 명령어 단위의 시뮬레이션이 가능한 시뮬레이터의 골격이 만들어지게 된다. 이 중에서 구현된 각각의 명령어들의 내용을 파이프라인의 각 단계에 따라서 나눈 후에 이들을 가지고 각 단계별로 모아서 구현해 주도록 하면 각 단계의 구현이 끝나게 된다.

여기에 다단 명령어의 경우를 처리해 주는 부분과 STALL 이나 FLUSH 같은 제어가 필요한 경우를 해결하게 된다면 본 시뮬레이터의 기본적인 구현이라고 할 수 있는 파이프라인 단계의 구현을 완료하게 된다.

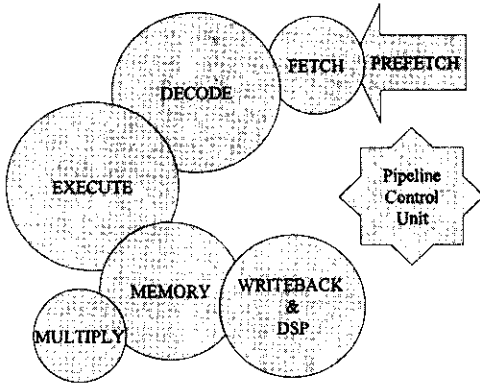

그림 2. 파이프라인의 동작 형태

3.2 파이프라인 제어부

각 파이프라인 단계의 구현이 끝나게 되면 이의 제어를 고려해야 하는데, 원활한 흐름을 위해서 그림 3.과 같은 순서의 제어가 필요하게 된다. 명령어의 단계별 수행은 이전 사이클에서 수행된 명령어를 다음 단계의 명령어 버퍼로 전달하는 방식으로 이루어지게 되는데 그 순서를 원래 파이프라인의 수행단계와는 역순으로 배열하게 된다. 이러한 순서의 제어는 소프트웨어 파이프라이닝과 같은 원리에 의해 실행된다고 볼 수 있는데, 이는 순차적인 실행에 의해 이전 명령어와 충돌이 일어나지 않도록 해 주는 것을 가능케 한다.

다음으로 파이프라인의 STALL과 FLUSH에 대한 제어가 필요해 지게 되는데, 대개의 경우 명령어의 요구에 의한 경우와 메모리 등의 자원의 사용이 중복될 경우에 일어나게 된다. 명령어의 경우에는 분기 명령이나 몇몇 메모리 제어 명령, 다단계 명령의 경우를 예로 들 수 있다.

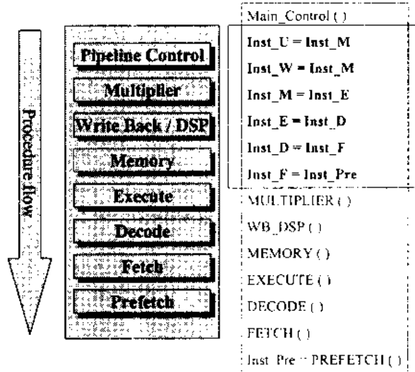

그림 3. 파이프라인 흐름 제어

우선 STALL 제어 방법을 LDC.L (load control register from memory)의 예를 들어 설명해 보면 아래그림 4와 같다. LDC.L은 두 사이클의 STALL 후 다음 명령어가 수행되도록 되어있다. 이를 구현하기 위해서는 DECODE 단에 LDC.L 명령어가 들어온 경우 FETCH 단에 들어갈 명령어를 두 사이클 동안 받아들이지 않고 이전 명령어를 그대로 유지할 수 있도록 해당 단계의 명령어 버퍼를 조작해 주고 DECODE 단에 다음 사이클과 그 다음 사이클에 입력되는 명령어는 수행치 못하도록 이 때의 DECODE 단의 명령어 버퍼에 NOP을 입력하도록 한다.

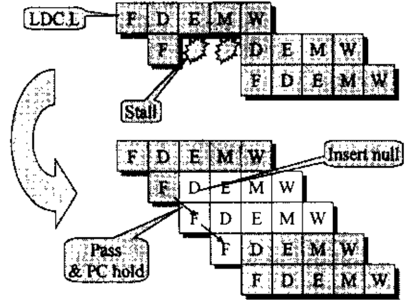

그림 4. STALL 신호의 제어 예

다음으로 FLUSH 제어 방법을 그림 5.에 나와있는 BT(branch if T is true)의 예를 들어 설명해 보면 다음과 같다. BT는 이 명령어 다음에 FETCH 단에서 오는 두 명령어를 버리고 다음에 분기 목적 주소에서 읽어 들인 명령어부터 정상 수행하도록 되어 있다. 이를 구현하기 위해서는 DECODE 단에 BT 명령어가 들어온 경우 DECODE 단에 들어갈 명령어를 두 사이클 동안 수행치 못하도록 이 때의 DECODE 단의 명령어 버퍼에 다음 두 사이클 동안 NOP을 입력하도록 한다.

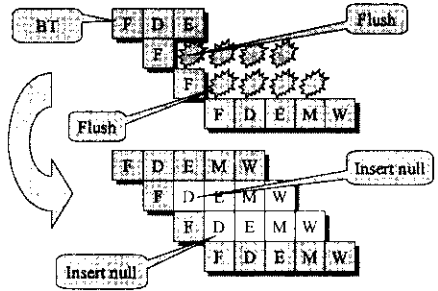

그림 5. FLUSH 신호의 제어 예

다음으로 다단계 명령의 제어 방법을 TST.B(test byte in memory)의 예를 들자면 그림 6과 같이 나타낼 수 있다. TST.B는 이 명령어 다음에 오는 명령을 두 사이클 동안 STALL 시키고 다단 명령의 각 단을 수행해 주게 되어 있다. 이의 구현을 위해서 DECODE 단에 들어온 TST.B 명령을 두 사이클 동안 BT 명령어가 들어온 경우 FETCH 단과 DECODE 단에 들어온 명령어들을 두 사이클 동안 고정 시켜두고 TST.B를 입력 받은 DECODE 단은 TST.B를 위한 스테이트머신을 돌린다. 세 사이클동안 TST.B라는 같은 명령어를 받게 되

는 EXECUTE 단에서는 이를 통해서 각 스테이트마다 마련된 해당 명령어에 대한 다단계 명령어 함수를 호출하여 수행토록 해 준다. 그리고 마지막 스테이트를 수행하게 될 경우에는 EXECUTE 단 마지막에서 스테이트 머신을 초기화 해 준다.

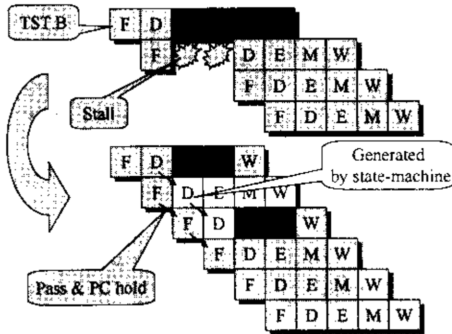

그림 6. 다단계 명령의 제어 예

3.3 그 외 부분의 구현

현재 구현된 시뮬레이터에 사용자와의 인터페이스를 구현하여 시뮬레이터의 동작을 제어할 수 있다. 따라서 현재 수행중인 시뮬레이터의 상태를 보이주는 모듈과 외부 입력에 따라서 시뮬레이터의 동작을 제어할 수 있도록 하는 모듈, 그리고 외부 파일로 되어 있는 메모리의 상태, 특히 ROM에 기록되어 있는 프로그램을 내부 메모리로 읽어 들일 수 있도록 해 주는 메모리 초기화 모듈을 필요로 하게 된다.

또한 앞서 완성된 파이프라인의 각 단계 중에서도 공통적으로 사용되는 부분들을 하나의 모듈로 만들어 주어 프로그램의 길이를 현저히 줄여줄 수 있다. 예를 들면 덧셈기나 곱셈기, 논리 연산 수행부, 메모리 제어부 등이 있다.

4. 구현 및 검증

본 연구에서 설계한 시뮬레이터는 Turbo C++ 3.0 상에서 구현하여 DOS 환경에서 동작하도록 구현되었다. 이의 테스트는 어셈블리 명령어들을 조합해서 만든 프로그램들을 바이너리 파일로 만들어 이를 ROM 파일로 만들어 시뮬레이터 상에서 로드하여 사용하도록 하였다.

이 구현된 시뮬레이터가 정상 동작한다는 것을 입증하기 위해서 우선 개별 명령어 단계에서의 테스트에서부터 다양한 DSP 알고리즘의 테스트의 순서로 이루어지게 되는데, 개별 명령어 테스트 시에는 각 파이프라인 단계에서 현재 수행중인 명령어가 정상적으로 그 동작을 수행하고 있는지를 단계별로 검사해 주고 다양한 알고리즘의 테스트 시에는 그 알고리즘을 제대로 수행하는지, 수행을 마친 후 그 결과가 정상적으로 나오게 되는지를 확인해 주었다. 그 중에서도 FIR 필터는 입력된 값과 보존된 이전 입력 값들을 필터계수와 곱하여 더해주는 동작을 하게 되는데 아래 그림은 구현된 시뮬레이터가 FIR 필터 프로그램의 동작 중에서 읽어 들인 계수들과 입력된 값들을 곱해서 더해주는 동작을 수행하는 모습이다.

이러한 테스트를 거쳐 원래 목적한 DSP 코어의 모델을 시뮬레이터로 올바르게 구현했다는 것을 입증 한 후에는 이 시뮬레이터와 구현된 HDL 모델의 동일성을 확인하기 위해, 앞서 테스트한 프로그램이 HDL 모델과 시뮬레이터에서 각 파이프라인 단계마다 동일한 동작을 하는지 메모리와 레지스터에 기록되는 값을 비교해 주게 된다.

이렇게 동일성이 입증된 시뮬레이터는 원래 모델의 성능향상을 위한 설계의 개선과 그 확인을 위해 사용하게 된다. 이러한 성능 평가를 위해서는 앞서의 테스트에 사용되었던 프로그램 보다는 좀 더 잘 알려진 성능 평가용 프로그램을 목적한 DSP 코어의 명령어들을 이용해 구현, 사용하게 된다. 설계 변경 후에 시뮬레이션에 걸리는 시간은 Verilog 모델을 통해 변경, 테스트 할 때보다 적은 시간을 필요로 하게 된다.

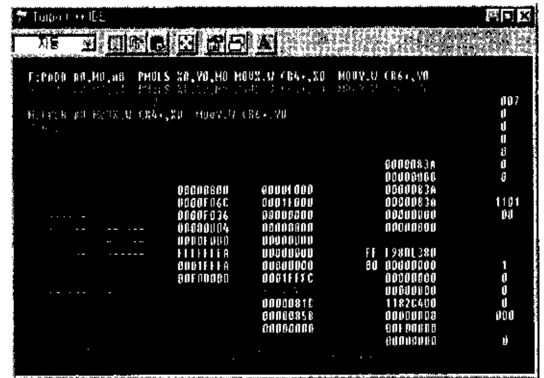

그림 7. 시뮬레이터로 FIR 동작을 수행하는 모습

5. 결 론

본 연구에서는 5 단계 파이프라인 구조를 갖는 DSP 코어를 위한 시뮬레이터를 설계하였다. 테스트 결과 실제 목표로 한 YS-DSP의 코어와 일부 하드웨어적으로 입력되는 외부신호 처리부를 제외한 나머지 부분에서 100%의 호환성을 가진다는 것을 확인하였고, 이를 바탕으로 YS-DSP의 성능향상을 위한 시뮬레이션에 걸리는 시간을 단축시킬 수 있었다. 사용되는 명령어들의 파이프라인 단계와 그 단계에서의 작업을 합리화하는 데에도 도움을 줄 수 있었으며 성능향상을 위한 변경된 설계의 테스트에 걸리는 시간을 단축시킬 수 있었다.

6. 참 고 문 헌

[1] Phil Lapsley, Jeff Bier, Amit Shoham, "DSP Processor Fundamentals" *by the Institute of Electrical and Electronics Engineers, Inc , 1997*

[2] Michael Dolle, Manfred Schlett, "A Cost-Effective RISC/DSP Microprocessor for Embedded Systems" *IEEE Micro, pp.32-40, Oct. 1995*