creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

비전시스템을 위한 DSP 개발

Development of a Custom DSP for a Vision System

2016년 2 월

서울대학교 대학원

전기정보 공학부

김 문 권

비전시스템을 위한 DSP 개발

지도 교수 백 윤 흥

이 논문을 공학석사 학위논문으로 제출함
2015년 12월

서울대학교 대학원
전기정보 공학부
김 문 권

김문권의 공학석사 학위논문을 인준함
2015년 12월

위 원 장 _____ 문 수 묵 _____ (인)

부위원장 _____ 백 윤 흥 _____ (인)

위 원 _____ 이 혁 재 _____ (인)

초 록

컴퓨터 비전 기술은 매우 많은 연산량을 필요로 하기 때문에 임베디드 환경에서는 프로세서의 성능 문제로 실시간 처리가 어렵다. 컴퓨터 비전 알고리즘의 높은 연산량과 실시간 처리 요건, 임베디드 시스템의 저전력요건을 만족하기 위해서는 알고리즘의 병렬성을 최대한 활용하여 특화된 하드웨어를 개발할 필요가 있다. 또한, 컴퓨터 비전에는 얻고자 하는 정보와 그 방법에 따라 수많은 알고리즘이 있고 새로운 알고리즘이 쏟아져 나오기 때문에 고정된 하드웨어블록으로 설계하기 보다는 프로그램 가능한 프로세서가 선호된다. 이렇게 특정 분야 응용들에 특화된 프로세서가 개발되면, 다중 프로세서 시스템-온-칩(MPSoC)에 포함하여 알고리즘의 일부, 특히 연산량이 많고 병렬성이 있는 부분을 특화 명령어를 통해 가속화 할 수 있다. 본 논문에서는 비전 시스템을 위한 DSP를 설계하고 개발하였다. 병렬연산이 가능한 저전력 VLIW 프로세서의 설계를 위해 SPD 툴을 사용하였고, 또한 SPD의 기능을 활용하여 LLVM 기반의 DSP 컴파일러를 개발하였다. Xilinx Zynq 보드에서 시스템을 구축해, 손 인식 알고리즘을 어플리케이션으로 하여 시스템을 검증하였다. DSP가 낮은 주파수에서 동작하는데도 불구하고 어플리케이션의 compute intensive 한 부분을 DSP에 실행한 결과는 ARM 프로세서에서 실행한 것보다 4배 정도 나은 성능을 보여 어플리케이션의 성능을 크게 가속화하였다.

주요어 : Computer Vision; VLIW Architecture; LLVM Compiler; Digital Signal Processing; Recognition
학 번 : 2014-22140

목 차

초 록 i
목 차 ii
제 1 장 서 론 1
제 2 장 background 와 관련 연구 3
제 1 절 Very Long Instruction Word(VLIW) Architecture3
제 2 절 Low Level Virtual Machine(LLVM) compiler 5
제 3 절 관련 연구 7
제 3 장 Implementation of custom DSP 9
제 1 절 DSP 개발 10
제 2 절 Application Specific Instruction Extension 14
제 3 절 Porting LLVM Compiler 19
제4절 메모리 최적화 24
제 5 절 Test and verification 26
제 4 장 실험 결과 29
제1절 실험 환경 30
제2절 실험 결과 32
제 5 장 Conclusions 35
참고 문헌 36
Abstract 38

표 목차

표 1. Experimental results 32

그림 목차

그림 1. normal compiler structure..... 5
그림 2. LLVM compiler structure..... 5
그림 3. Processor Designer 기능 10
그림 4. Non-pipelined 방법..... 12
그림 5. Internal design of DSP architecture 13
그림 6. 손 인식 알고리즘 GenerateFC함수..... 14
그림 7. Internal design of DSP architecture 15
그림 8. SIMD ALU 연산 구현..... 16
그림 9. SIMD Compare 연산 구현..... 17
그림 10. find first bit..... 18
그림 11. LLVM IR to Machine Code Mapping..... 19
그림 12. Processor Designer tool chain 21
그림 13. Intrinsic function 사용 예..... 22
그림 14. Sanity test..... 26
그림 15. printf 실행 과정..... 27
그림 16. Layout of the prototype system..... 30
그림 17. Result of hand recognition 33
그림 18. 손 인식 실험 결과..... 34

제 1 장 서 론

컴퓨터 비전 기술은 매우 많은 연산량을 필요로 하기 때문에 강력한 워크스테이션에서만 황용을 하였으나, 요즘에 와서는 모바일영역에서도 많이 활용하고 있다. 이것은 전력 소모 제한에도 불구하고 오늘날 모바일 프로세서의 프로세싱능력을 크게 향상시킬 수 있기 때문이다. 또한 컴퓨터 비전기술을 모바일영역에 적용하면 많은 흥미로운 어플리케이션을 사용할 수 있게 된다. 허나 여전히 비전 어플리케이션의 많은 연산량 요구로 의해 여러 가지 문제를 겪고 있다. 사용자에게 윤활한 사용체험을 제공하기 위해, 1초에 이미지의 수십 개 frame을 처리해야 하고, 또 분석해서 유용한 정보들을 그 이미지로부터 적출해야 한다. 이를 해결하기 위한 유효한 해결방법 중 한나는 high-end 범용 프로세서를 컨트롤 프로세서로 사용하고 특화된 하드웨어를 이용해 알고리즘의 일부, 특히 연산량이 많은 부분을 가속화 하는 것이 있다. 이 방법은 이미 신호처리영역에서 많이 사용되고 있다.

다른 한 부류의 비전 어플리케이션은 알고리즘이 고정되지 않은 것이다. 비록 SIFT[1], HOG[2]와 같은 많이 사용되는 알고리즘이 있지만 아직 업계의 표준으로 사용하기에는 부족하다. 이 외에도 수많은 어플리케이션과 그 알고리즘들의 변종들이 있다. 만약 고정된 하드웨어 블록을 사용하면 성능기준에는 만족시킬 수 있지만, 다른 비전 어플리케이션을 그 하드웨어에 사용할 수 없을 뿐만 아니라, 업데이트도 불가능하고 그 알고리즘을 특수 환경에 적용하는 것도 불가능하게 될 것이다.

이를 해결하기 위해 고정된 하드웨어 블록을 설계하기 보다는 프로그램 가능한 프로세서가 선호된다. 이렇게 특정 분야 응용들에 특화된 프로세서가 개발되면, 다중 프로세서 시스템-온-침(Multi-Processor System-on-Chip)에 포함하여 특화 명령어를 추가해 특수 목적으로 사용할 수 있다. 또한, Light-weight 범용프로세서를 control-oriented 코드를 실행하는 것과 그 프로그램 가능한 특화 프로세서를 컨트롤하는데 사용할 수 있다. 그리고 프로그램 가능한 특화 프로세서는 어플리케이션의 일부, 특히 연산량이 많은 부분을 가속화 할 수 있다. 현재 high-density 칩 기술의 덕분에, 우리는 여러 개 프로세서를 한 개

칩에 넣어 이 컴포넌트들간의 통신 오버헤드를 최소화 할 수 있다.

본 논문에서는 비전 시스템을 위해 custom DSP를 설계하고 개발하였다. 병렬연산이 가능한 저전력 VLIW 프로세서의 설계를 위해 SPD(Synopsys Processor Designer[3]) 툴을 사용하였고, 또한 SPD의 기능을 활용하여 LLVM(Low Level Virtual Machine [4]) 기반의 DSP 컴파일러를 개발하였다. 손 인식 알고리즘을 어플리케이션으로 우리 시스템에서 실행하여 DSP를 검증하였다. Prototype 시스템은 Xilinx Zynq evaluation board[5] 에서 구축하였고, ARM 프로세서를 컨트롤 프로세서로 사용하였다. 컨트롤 프로세서와 DSP간의 데이터 통신 량이 크기 때문에 DMA제어기를 이용하여 메모리간에 데이터를 전송하도록 하였다

제 2 장 background 와 관련 연구

제 1 절 Very Long Instruction Word(VLIW) Architecture

최근의 프로세스들은 보다 향상된 성능을 얻기 위해 여러 개의 Functional unit을 두어 여러 개의 instruction을 동시에 실행하고자 한다. 이런 아키텍쳐에서는 conflict와 hazard가 없는 여러 개의 instruction을 병렬적으로 실행할 수 있는데 이와 같은 parallelism을 ILP (instruction level parallelism) 이라고 한다. 성능을 향상시키기 위해서는 가능한 많이 ILP를 활용하여야 한다. ILP를 이용한 대표적 아키텍쳐에는 superscalar architecture와 VLIW architecture가 있다.

이 논문에서 제안하는 아키텍쳐는 VLIW architecture의 형태를 가지고 있다. VLIW는 Very Long Instruction Word의 약자로 기존의 단일 Pipelined data path와 Functional unit을 가지는 processor의 최대 IPC(instruction Per Cycle) 가 1임을 극복하기 위해 나온 여러 형태의 processor들 중 하나이다[10]. 기존의 단일 Functional unit과 Instruction fetch width를 가지고 있는 Processor들과는 달리 VLIW는 여러 개의 Functional unit과 넓은 Instruction fetch width를 가지고 있는데, 이는 기본적으로 한꺼번에 같이 수행될 수 있는 여러 Instruction을 한 번에 읽어와 동시에 실행하자는 생각을 바탕으로 한다. 이를 통해 IPC를 1 이상으로 높일 수 있게 되는 것인데 이러한 성능의 증가는 증가된 Instruction Memory의 크기와 Read port의 width, 추가된 Functional unit 및 Data path logic과 Pipeline register에 대한 보상이다.

superscalar architecture에서는 instruction을 memory에서 가져온후 hardware가 어느 instruction들이 resource conflicts와 data dependency 없이 병렬적으로 실행할 수 있는지를 결정한다. Resource conflict는 하나 이상의 instruction이 동일한 hardware component 사용을 요구하였을 경우 발생한다. Data dependency hazard는 하나의 instruction 수행이 다른 instruction의 실행 결과를 필요로 하는데 필요

되는 instruction이 아직 실행 완료 되지 않은 경우에 발생한다. 일반적으로 superscalar architecture에서는 hazard를 발견하기 위해 복잡한 hardware 디스패칭 로직이 요구 된다.

하지만 VLIW architecture에는 복잡한 hardware 디스패칭 로직이 없다. VLIW는 고정된 개수의 instruction을 가져와 곧 바로 다음 pipeline functional unit에 이슈 한다. 그리고 정확한 실행을 보장하기 위한 hazard detection은 software approach로 해결한다. 여기서 compiler는 resource conflict와 data dependency hazard가 발생하지 않게끔 분석하고 scheduling한다.

VLIW architecture가 복잡한 hardware 디스패칭 로직이 필요하지 않기 때문에 칩의 크기나 파워소모에 제한 요구가 많은 임베디드 시스템에 환영을 받고 있다. VLIW architecture은 superscalar architecture 에 비해 보다 적은 코스트, 작은 회로와 파워 소모량, 그리고 보다 작은 hardware 디자인으로 ILP를 이용할 수 있다. 반면, instruction scheduling을 맡고 있는 compiler가 보다 복잡해 지고 중요해 진다.

제 2 절 Low Level Virtual Machine(LLVM) compiler

LLVM은 2000년에 the University of Illinois at Urbana-Champaign에서 시작한 프로젝트로서 원래는 Low Level Virtual Machine의 약자를 가리켰지만, LLVM이 성장하면서 다양한 목적을 가지게 되면서 현재는 이 약자는 사용하지 않는다. 지금 LLVM은 optimizer, code generator, assembler 등을 포함한 오픈 소스 compiler toolchain 이다.

GCC와 달리 LLVM은 C++로 쓰여 졌고 재사용 가능한 libraries와 모듈 API들로 구성 되였다. 임의의 프로그래밍 언어의 정적 및 동적 편집을 모두 지원할 수 있는 효율적인 컴파일러 인프라를 제공하는 것을 목표로 하기 위해 컴파일러 제작은 독립적인 frontend, LLVM optimizer, 독립적인 backend 3개 별도의 단계를 거친다. 따라서 새로운 타겟을 추가하려면 frontend와 optimizer을 수정하지 않고 단지 backend만 수정하는 것으로 완성할 수 있다.

그림 1. normal compiler structure

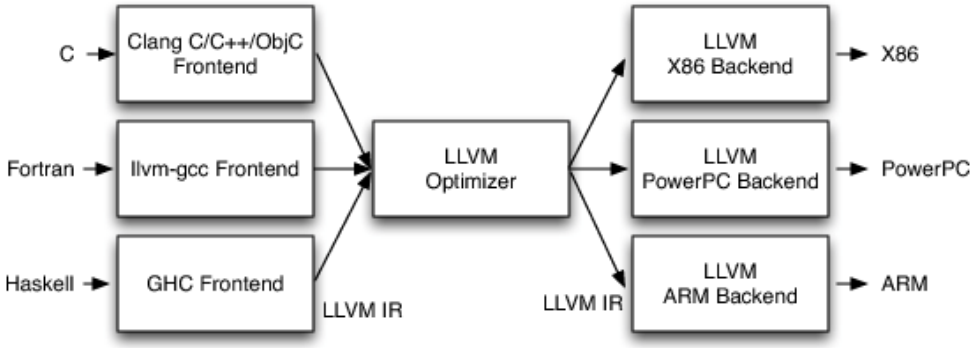

그림 2. LLVM compiler structure

현재 LLVM 기본구조는 넓은 영역에서 사용되고 있다. 애플은 이미 2005년부터 Mas OS x 및 iOS 개발에 LLVM 을 중요한 툴의 하나로 사용하였고 LLVM의 frontend인 Clang compiler은 SDK Play Station 4 console에 사용 되었다. Clang은 GCC보다 3배 빠른 속도로 Objective-C를 컴파일할 수 있었다

제 3 절 관련 연구

컴퓨터 비전은 최근에 주목 받는 기술로 감시, 로봇공학, 수화 통역, 인간-컴퓨터 상호작용, 산업용, 자동차 안전 등 다양한 분야에 적용되고 있다. 동작 인식을 위해서는 영상 데이터를 가지고 다양한 연산을 수행하여야 하는데, 먼저 의미 있는 정보를 해석해 내기 쉽게 하기 위한 이미지 변환, 필터링 등이 수행되고, 다음으로 동작 인식의 대상이 포함된 구역을 구별한 다음 해당 구역에 대해 대상을 인식하고 모양 및 동작을 판단하는 것이 일반적인 과정이다. 이런 과정은 카메라에서 영상정보를 USB나 FireWire(1392), Ethernet 등을 통해 PC 등의 메인 컴퓨터에 입력 받아 처리하는 것이 일반적이나, 영상처리의 일부 또는 전 과정을 카메라 단에서 수행하여 그 결과만을 메인 컴퓨터에 전송하는 방식의 시스템도 있다. 이러한 시스템은 영상처리가 되기 전의 데이터를 전송하는 데에 필요한 bandwidth가 상당하며, 영상의 화질이 좋아질수록 bandwidth가 커진다는 점, 그리고 임베디드 시스템에서는 범용 프로세서나 GPU 등을 사용하는 것보다 어플리케이션에 특화된 프로세서를 카메라 단에 달아서 사용하는 것이 가격 및 전력 면에서 효과적일 수 있다는 점에서 장점을 갖는다. 동작 인식을 카메라 단에서 수행하고 그 결과만을 메인 컴퓨터에 전송한다면 전송해야 할 데이터의 양은 굉장히 줄어든다. 이러한 시스템은 메인 컴퓨터에서 굳이 영상 데이터 전체가 필요한 것이 아니고 동작 인식 결과만 필요하거나, 카메라와 센서 네트워크를 결합시킨 무선 카메라 네트워크, 또는 다양한 임베디드 시스템에 적용할 수 있을 것이다.

카메라 단에서 기본적인 영상처리를 수행하는 것은 Smart Camera라는 명칭으로 꽤 오래 전부터 학계에서 언급되어 왔다. 다만, 동작 인식을 카메라에 달린 프로세서에서 수행하는 것은 비교적 최근에 논문이 나오고 있다. 그 중 상당수는 FPGA[6][7]를 바탕으로 알고리즘을 돌려보기 위한 것이고, FPGA를 이용한 reconfigurability를 장점으로 내세운 논문도 있으나 FPGA는 굉장히 세밀한 reconfiguration이 가능한 반면 최적화된 성능을 내지는 못한다는 단점이 있다. TI 등 기존의 DSP를 사용하여 동작 인식 시스템을 구성한 것도 있지만, DSP가 동작 인식에 특화되어 개발된 것이 아니라

전반적인 영상 처리를 위한 프로세서라는 점을 감안하면, 동작 인식만을 위해 최적화된 솔루션에는 미치지 못할 것이다.

산업계에서는 굉장히 많은 업체가 동작 인식 산업에 진출해 있다. 대표적으로 Play Station 4의 동작 인식 Depth Sensor와 개발용 API를 제공하고 있는 SoftKinetic 사와 MS Kinect의 첫 버전의 센서와 개발 플랫폼을 제공해준 PrimeSense 사가 있다. PrimeSense 사는 2013년 말 Apple에 인수된 바 있다. Qualcomm, Intel 등도 이 분야에 투자를 많이 하고 있다. 그 외에 GestureTek, eyeSight, PointGrab 등 많은 회사가 Software 기반의 솔루션을 제공하고 있다.

위에 언급한 업체는 Software를 기반으로 하여 동작 인식 솔루션을 제공하고 있기 때문에 특별한 하드웨어가 없이 카메라와 PC만 있으면 적용 가능하다는 장점이 있지만 임베디드 시스템에 적용하기에는 전력소모나 가격의 측면에서는 단점이 있다. 반면 Cognivue 사는 SIMD array 구조를 갖는 자체 코어를 사용하여 저전력 고성능의 솔루션을 제공하고 있고, 이 점을 마케팅에도 강조하고 있다.

비전 처리분야에는 hardware를 이용한 연구가 굉장히 많이 진행되고 있다. 그 중에서 FPGA를 이용하여 hand gesture recognition을 가속하는 것이 있다. FPGA는 prototyping을 가속하는데 적합하지만 면적이나 속도에서 최적화된 선택은 아니다. 또한 GPU의 대규모 병렬처리 능력을 이용해 비전 어플리케이션을 가속화 하는 노력도 있다[8]. 그러나 GPU는 parallelism 유형에 제한되어 있고 또 embedded system에 있어 비용이 너무 크다.

그리고 다른 연구들에서는 DSP를 사용한 것이 있지만 customizable 하지 않고 SoC에 통합시킬 수 없다. Bone-V[9]는 custom 칩으로서 뛰어난 성능을 나타냈지만 오직 object recognition만 적용된다. 산업계에서도 요즘 비전 프로세서를 개발하여 비전시스템에 통합하는데 큰 관심을 보여주고 있다. 우리의 연구들과 비슷한 방향으로 연구되는 것들에는 IMAPCAR2[11], EyeQ2[12], Myriad 2[13] 등이 있다.

제 3 장 Implementation of custom DSP

우리가 제안한 DSP는 Synopsys사의 Processor Designer툴을 사용하여 VLIW architecture를 기반으로 하여 개발하였다. VLIW 는 여러 개의 instruction을 동시에 수행하되, instruction의 스케쥴링은 컴파일러에 의존하여 하드웨어의 복잡도를 줄임으로써 비용과 전력소모를 최소화한 아키텍쳐이다. C코드로 된 어플리케이션을 우리의 DSP에서 돌리기 위해 Synopsys사의 Processor Designer툴의 LLVM Compiler Designer 기능을 이용하여 컴파일러를 개발하였다. 컴파일러 및 프로세서의 검증을 위해 test code 들을 GCC 컴파일러로 컴파일하여 Processor Designer 툴을 통해 프로세서 시뮬레이터로 돌린 결과와 비교하였다. 또한 Processor Designer로 생성한 RTL을 modelsim을 이용하여 검증하였다.

이 장의 첫 번째 절에서는 Processor Designer툴을 사용해 DSP를 개발하는 과정에 대해 소개할 것이고, 두 전째 절에서는 어플리케이션의 가속을 위하여 추가한 특화 instruction에 대해 설명할 것이며, 세 번째 절에서는 컴파일러의 개발과정에 대해 다루어볼 것이다. 메모리 최적화에 대한 내용은 4장에서 소개될 것이고, 마지막 절에서는 테스트과정에 대해 알아볼 것이다.

제 1 절 DSP 개발

본 논문에서는 Synopsys 사의 Processor Designer 툴을 사용하여 DSP를 개발하였다. Synopsys Processor Designer은 Verilog와 같은 HDL(Hardware Description Language) 보다 높은 레벨의 프로세서 기술 언어를 사용하여 ASIP (Application Specific Instruction-set Processor)를 쉽게 만들 수 있는 환경을 제공하여 주는 툴이다. 이 프로세서 기술 언어를 LISA(Language for Instruction Set Architecture) 라고 한다. LISA를 통해 우리는 프로세서 pipeline stage와 각 pipeline에서 어떤 동작을 할 것인지를 기술하면 Compiler, Assembler, Linker, Simulator, Debugger를 아우르는 SDK(software development toolkits)를 생생해 주고, 합성 가능한 RTL code 또한 생생해 준다(그림 3). 하지만 LLVM 컴파일러의 생성은 자동적으로 완전한 컴파일러가 생성되는 것이 아니고, 그 과정을 좀더 간단히 할 수 있도록 도와주는 정도이다. 비록 Processor Designer에 의해 생성된 RTL코드는 프로그래머가 코딩 한 RTL처럼 최적화 되어 있지는 않지만 적용하기에는 문제가 없다. 또한 high-level 디자인이 가능하고 자동적으로 생성된 SDK toolchain에 의해 우리는 보다 짧은 시간에 prototype 시스템을 만들어 그의 여러 개 특징을 테스트하여 최종 디자인을 결정한 후 RTL 코드를 수동적으로 최적화할 수도 있다.

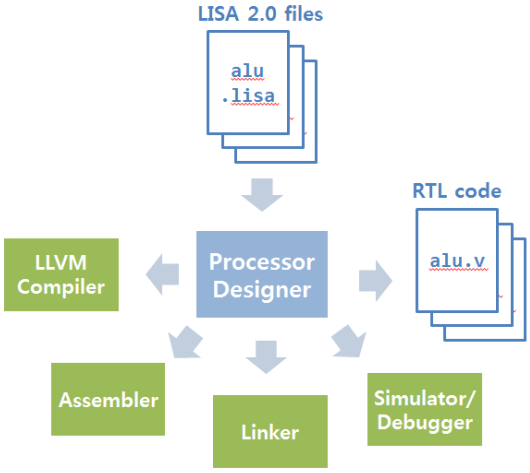

그림 3. Processor Designer 기능

우리의 custom DSP는 VLIW 아키텍쳐를 기반으로 하였다. VLIW는 instruction을 병렬로 처리할 수 있지만, 하드웨어에서 instruction들의 의존성 분석 및 동작 스케쥴링을 지원하지는 않고, 컴파일러에 의존하는 아키텍쳐이다. 따라서 하드웨어가 굉장히 간단하여 저전력 임베디드 시스템에 적합한 구조이다. Computationally intensive 작업은 보통 predictable 행위가 있기 때문에 복잡한 하드웨어 구성이 없다고 해서 성능에 큰 영향을 주지 않는다. 그리고 다른 한편으로는, 전력소모와 실리콘 면적을 굉장히 많이 줄일 수 있다. 바로 이러한 원인으로 인해 지금 많은 DSP(TI's C6000 DSPs[14], Trimedia[15], SHARC[16], 또 최근 SHAVE processors 에는 Movidius Myriad[17] 비전 칩을 포함하였다)에서는 VLIW 아키텍쳐를 사용하고 있다. VLIW 프로세서는 최적화 된 코드를 생성할 수 있는 강력한 컴파일러를 필요로 하고 있는데 Processor Designer이 LLVM backend stub 코드를 제공할 수 있기 때문에 우리는 많은 작업을 하지 않고도 최적화 된 코드를 얻을 수 있다.

우리는 Processor Designer package에서 가장 기본적인 VLIW아키텍쳐를 설계한 후 Processor Designer 툴을 이용하여 일부 기능을 추가하거나 수정을 했다. 그 중에서 가장 크게 변동된 부분은 asynchronous 메모리를 synchronous 메모리로 수정하는 것이다. 기본 모델에는 메모리가 비동기적으로 되어 있어 DSP가 메모리 주소를 가져오는 것과 데이터메모리에서 데이터를 가져오는 것이 한 cycle 에 완성되었다. 그러나 우리가 개발한 시스템에서는 SRAM을 사용하기에 동기적으로 되어 있다. 이 문제를 해결하기 위해서는 non-pipelined와 pipelined 두 가지 방법이 있다.

Non-pipelined 방법을 Load instruction로 애를 들어 설명하겠다. 그림 4는 Load instruction의 pipeline Execute stage의 LISA코드중 한 단락이다. Load instruction가 실행할 때 OUT.stalled() 가 false이므로 data_mem.setAddress_c0가 콜 되어 메모리 주소를 가져오게 된다. 그리고 pipeline stage가 OUT.stall에 의해 멈추게 된다. 그 후 다음 cycle에 프로세서는 Load instruction의 Execute stage를 다시 한번 실행하게 되는데 DATA_MEM_RD_LATENCY를 1로 설정했을 경우 OUT.stalled가 true 이므로 프로세서는 data_mem.getData_c0 통해 data를 메모리에서 가져오게 된다. 이 방법을 사용하면 memory를

비동기적으로 하였지만 매번 Load instruction을 실행 할 때뿐만 아니라 매번 code 메모리에서 instruction들을 가져올 때도 한 cycle을 낭비하게 되어 성능에 큰 영향을 끼치게 되어 보통 pipeline적인 방법을 사용하게 된다.

우리의 DSP에 적용한 Pipeline적인 방법은 주소를 가져오는 것과 데이터를 메모리에서 가져오는 것을 서로 다른 Pipeline에 두어 진행하는 것이다. 역시 Load instruction을 예로 들면 data_mem.setAddress_c0를 pipeline의 Execute stage에 두고 data_mem.getData_c0 pipeline의 Memory stage에 넣어 둔다. 개발한 프로세서가 instruction을 code 메모리에서 가져 오는 것도 비동기적으로 하기 위해 우리는 pipeline을 원래의 5개 stage(Fetch, Decode, Execute, Memory, Writeback)에서 6개 stage(Prefetch, Fetch, Decode, Execute, Memory, Writeback)로 변경 하였다(그림 5). LISA코드에서 instruction 주소를 가져오는 prog_mem.setAddress_r() 함수를 Fetch stage에서 추가된 stage인 Prefetch stage에 넣어 instruction을 가져오는 함수 prog_mem.getData_r()와 분리시킨다. 이런 방법을 사용함으로써 DSP가 cycle낭비 없이 비동기적 메모리를 적용할 수 있게 하였다.

```
if (!(OUT.stalled())) {
  data_mem.setAddress_c0(addr);
  LDW_CNT = DATA_MEM_RD_LATENCY - 1;
  OUT.stall();
} else if (LDW_CNT > 0) {
  LDW_CNT--;
  OUT.stall();
} else {
  data_mem.getData_c0(value);
  OUT.writeback_value = value.to_uint32();
}
```

그림 4. Non-pipelined 방법

또한 실리콘 면적과 성능에서도 아키텍쳐를 최적화하였는데 구체적으로 forward branch decision logic을 decode stage로 이동시켜 conditional branch penalty를 줄인 것, zero-register을 할당 가능한 register로 터닝한 것, 사용하지 않는 instruction들을 삭제한 것, 그리고 재사용 가능한 logic을 최대한 재활용한 것과 multiplication과 같은

아주 적게 사용하지만 꼭 필요한 instruction을 VLIW 의 한 slot에만 수행하게 하는 것이다.

그림 5는 DSP 아키텍쳐 구조를 개략적으로 나타낸 것이다. 간략하게 표현하기 위해 Bypassing회로와 branch회로는 생략하였다. 모두 4개의 슬롯이 있어 최대 4개의 instruction을 한번에 수행할 수 있다. 기본적인 ALU, CMP 연산은 모든 슬롯에서 지원되고, Multiply 연산이 하나의 슬롯에서 지원이 되며, 뒤에 언급될 SIMD연산과 특화 명령어가 VFU로 표시된 두 개의 슬롯에서 지원이 된다.

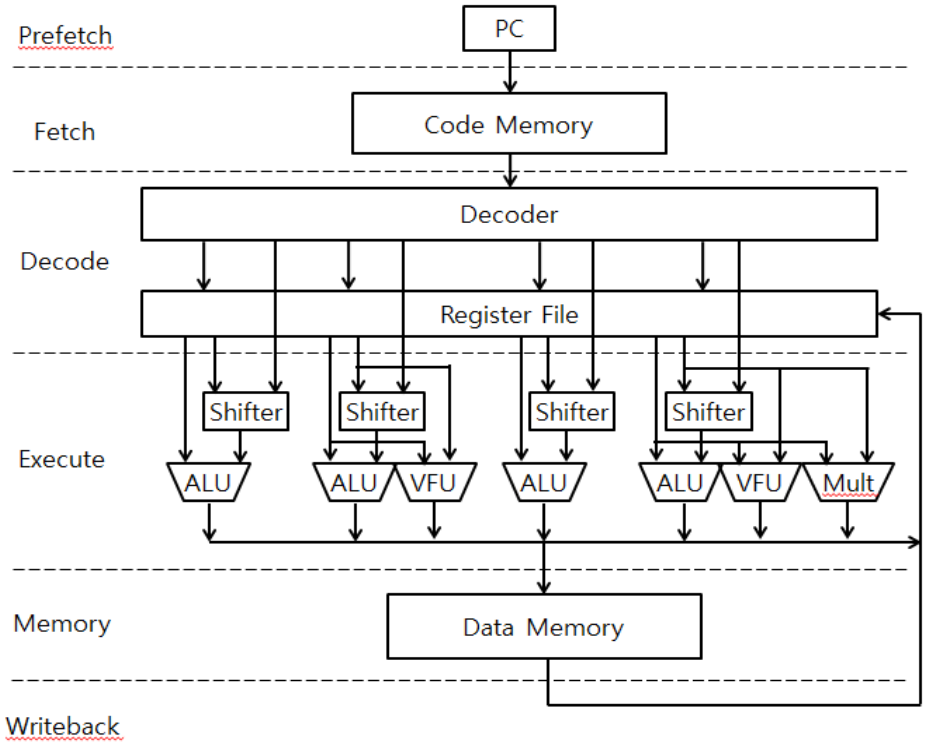

그림 5. Internal design of DSP architecture

제 2 절 Application Specific Instruction Extension

제1절에서 보인 바와 같이 제안된 architecture는 vision application에서 상대적으로 많이 나오는 병렬연산의 동작에 있어 기존의 VLIW architecture보다 나은 성능을 낼 수 있도록 최적화를 하였다. 이를 위해 해당 application을 분석하여 자주 나오는 패턴의 instruction을 빠르게 수행할 수 있는 방안을 모색하였고, 이번 장에서는 그 instruction들에 대해 설명을 하도록 할 것이다.

3.2.1 Algorithm 분석

본 논문에서 사용할 Application은 손 인식 알고리즘이다. 효율적인 DSP활용을 위해 알고리즘을 분석해 본 결과, GenerateFC이란 gradient를 계산하여 edge들을 찾는 부분이 계산량이 가장 많았다. GenerateFC 함수는 이미지의 매 라인에 대해 한번씩 수행하는데, 주로 Paired edge 리스트를 찾는 함수, 앞뒤 라인의 Paired edge들을 연결가능한지를 확인하는 함수, 그리고 finger candidate 리스트를 만들고 분석하여 그들을 업데이트하는 함수들로 구성되었다(그림 6). 위 함수가 병렬성이 높으면서 계산량이 가장 많아 이 부분을 DSP에서 수행하도록 하였고 여기서는 그 부분에 중점을 두고 설명할 것이다.

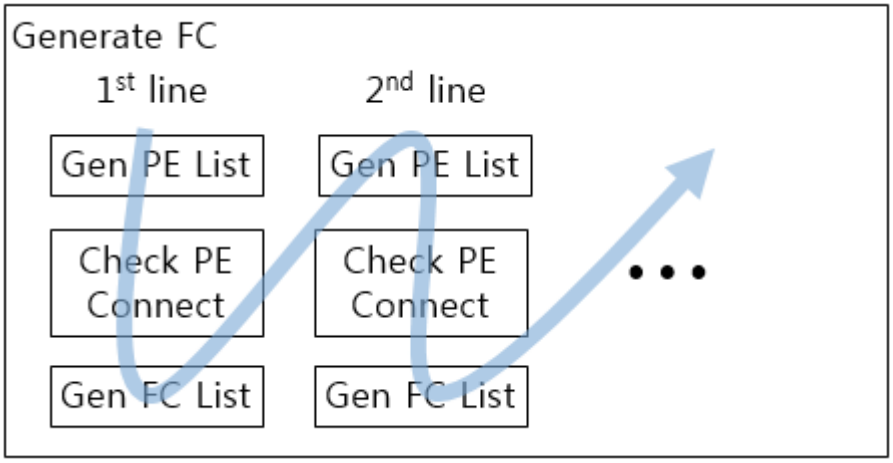

그림 6. 손 인식 알고리즘 GenerateFC함수

손 인식 알고리즘의 GenerateFC함수에서 시간을 가장 많이 소모하는 부분을 살펴보면, 이미지 frame의 매 pixel를 매 라인마다 왼쪽에서 오른쪽으로 스캔하여 손가락의 edge가 어디 있는지를 찾아내는 것이다. 그리고 찾아낸 edge x 포지션의 라인, 위치 등 정보를 저장하는 것인데, 이것을 그림 7로 간단하게 표현하였다. 알고리즘 자체는 간단하지만 이미지의 매개 pixel마다 많은 연산이 필요해서 시간을 매우 많이 소모한다. 그러나 연산이 규칙적이고 또 병렬적으로 진행할 수 있는 부분이 많아 우리의 DSP에서 돌리는 것이 매우 적합하였다.

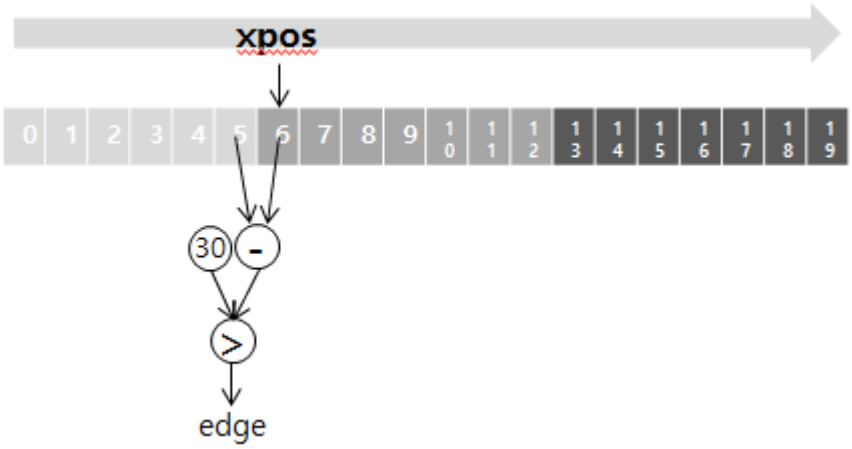

그림 7. Internal design of DSP architecture

3.2.2 SIMD(Single Instruction Multiple Data) instruction

우리는 특화 명령어를 이 논문에서 설명할 application 외에 향후 추가될 수 있는 다양한 vision application 에서도 공통으로 쓰일 수 있도록 최대한 general하게 만들었다. 여기서는 간단하게 손 인식 알고리즘에서 시간을 가장 많이 소모하는 부분을 살펴보고 분석한 결과 어떻게 하드웨어 설계를 개선하였는지에 대해 설명할 것이다.

이미지의 edge를 찾기 위해서는 서로 근접한 pixel들의 gradient를 계산하여야 하는데 이 부분에는 병렬성이 높아 병렬적으로 실행할 수 있었다. 이미지가 한 pixel당 8bit(1 byte)로 표현하기 때문에 32-bit data path를 가지고 있는 DSP에서 한 pixel씩 연산을 수행하는 것은

자원 낭비였다. 따라서 우리는 네 개의 pixel에 대해 동시에 연산이 가능한 Vector-Vector ALU 연산을 추가하였다. 이러한 어프로치를 SIMD(Single Instruction Multiple Data) 라고 하는데 GPU와 같은 병렬처리기기의 기본구조이다. 이와 같은 어프로치는 다른 어플리케이션에서도 많이 사용할 수 있어 우리는 SIMD 명령어를 우리의 DSP에 추가하기로 하였다.

이 클래스의 instruction들의 구체적인 동작은 아래와 같다. C코드가 Vector 연산을 지지하지 않지만, 수정한 컴파일러의 지원을 받아 프로세서는 4개의 pixel 데이터를 하나의 32bit width register에 넣을 수 있었다. 그 후, 프로세서는 그림 8과 같이 LISA의 to_ubitvec() 함수에 의해 실현된 하드웨어의 도움으로 32bit의 데이터를 4개의 8bit Vector 데이터로 전환 한다. 4개의 Vector-Vector ALU 연산 결과는 또 다시 LISA의 assign()함수에 의해 실현된 하드웨어의 도움으로 하나의 32bit width register에 저장함으로써 4개의 ALU연산을 하나의 instruction으로 완성할 수 있게 되었다.

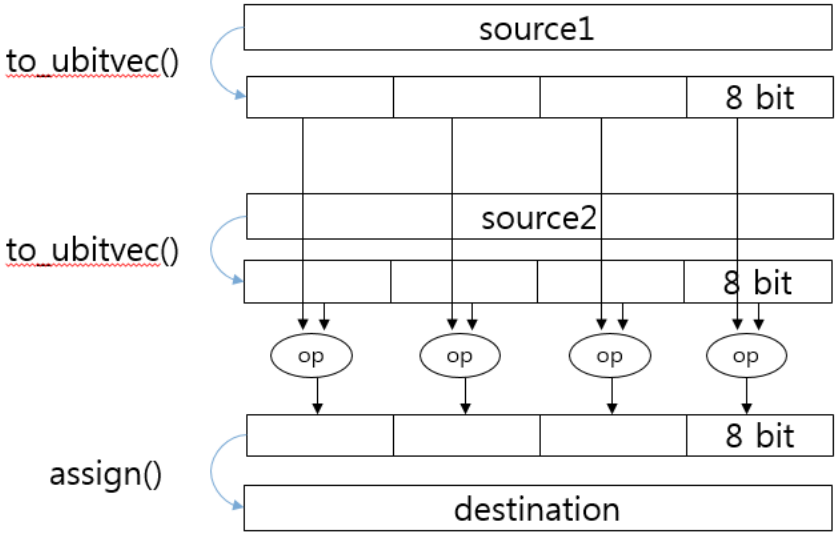

그림 8. SIMD ALU 연산 구현

또한 Vector-Scalar Compare 연산을 추가하여, edge detection을 빠르게 수행할 수 있도록 하였다(그림 9). 알고리즘을 살펴보면 근접한 Pixel의 gradient를 계산한 후 그 결과를 일정 값과 비교하여 edge인지를 판단한다. 이 하드웨어를 사용함으로써 4번의 비교 연산을

하나의 instruction으로 완성할 수 있다. 우리의 custom DSP에는 지금의 손 인식 알고리즘에 필요한 Vector-Vector ALU, Vector-Scalar Compare 연산만 추가하였지만, application에 따라 Vector-Vector Compare, Vector-Scalar ALU등 다양한 SIMD 연산도 간단히 추가가 가능하다. 물론 이러한 기능을 하드웨어 프로그래머가 HDL로 직접 구술하여 실현할 수 있겠지만 LISA를 통해 구현 시, 단 한 개의 함수를 통해 그를 표현할 수 있다는 것이 매우 간단하고 편리하다.

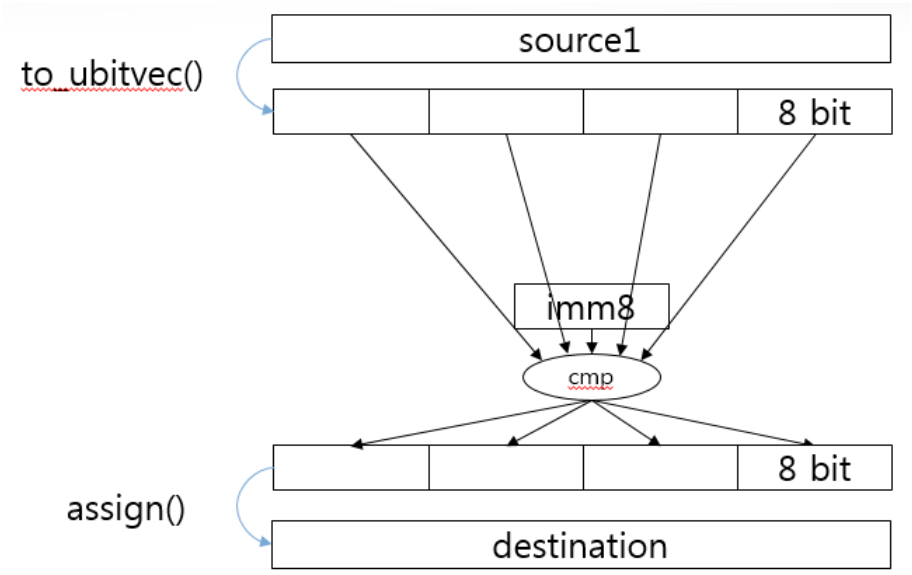

그림 9. SIMD Compare 연산 구현

3.2.3 Find bit instruction

위에서 SIMD ALU 연산과 SIMD Compare 연산을 한 후, 네 개의 결과가 한 register에 저장되어 있어 우리는 그 중에 edge가 있는지, 있으면 어느 포지션에 있는지 알아 내야 한다. 이 부분의 연산은 엄격하게 병렬되어 있지 않아 가속화 하는데 있어 쉬운 일은 아니다. 이것을 해결하기 위해 추가한 특화 명령어는 register에 들어있는 한 word에 대해 가장 첫 번째로 1로 셋팅이 된 bit의 포지션을 구하는 것이다(그림 10). 하지만 앞에서 언급한 Vector-Scalar Compare 계산을 한다 해도 하나의 32bit width register에 4개의 Boolean value 밖에 들어 있지 않아 이 특화된 명령어를 추가했다 하여 얻은 이득은

그다지 많지 않다. 하여 우리는 앞의 SIMD Compare instruction을 조금 수정하여 한 bit 마다 하나의 비교 결과를 가질 수 있게 함으로 8개의 Vector-Scalar Compare 결과를 하나의 32 bit register로 표현할 수 있게 하였다. 그러면 만약 연속된 32개의 pixel에 edge가 하나도 없다고 하면, 이것을 체크하는데 오직 1개의 instruction만으로 실행 할 수 있다. 이를 통해 앞서 detect된 edge의 위치를 빠르게 구할 수 있다. 이러한 연산은 일부 상업용 DSP에도 흔히 지원되는 명령이다.

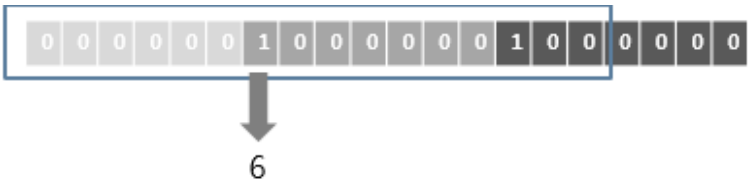

그림 10. find first bit

제 3 절 Porting LLVM Compiler

Processor Designer은 또한 개발한 프로세서를 위해 LLVM 컴파일러 backend를 생성할 수 있다. 허나 그것은 완전히 자동적 인 것은 아니고 프로그래머들이 일부 정보들을 Processor description 부분에 추가해야 한다. 그리고 생성된 LLVM 코드를 문제가 없는지 체크하고 또 일부 코드는 직접 추가해야 한다.

3.3.1 LLVM IR to Machine Code Mapping

Compiler는 보통 C/C++/Java 등 프로그래밍 언어로 짠 코드를 target architecture에 독립적인 중간 표현(IR: Intermediate Representation)으로 바꾸는 부분과 IR을 target architecture의 machine code로 바꾸는 부분으로 나뉜다. 우리가 개발하려는 DSP에 맞는 컴파일러를 만들기 위해서는 LLVM IR을 machine code로 바꾸어 주는 부분을 구현하면 된다. 이를 위해 그림 11에 나타낸 바와 같이 LLVM IR의 각 명령어를 target architecture의 적당한 명령어로 mapping을 해주는 것이 필요하다. LLVM IR의 명령어의 의미를 파악하고, LISA로 기술 된 target architecture의 각 명령어의 의미와 syntax를 비교하여 적당한 명령어를 선택하여 mapping을 해줘야 한다.

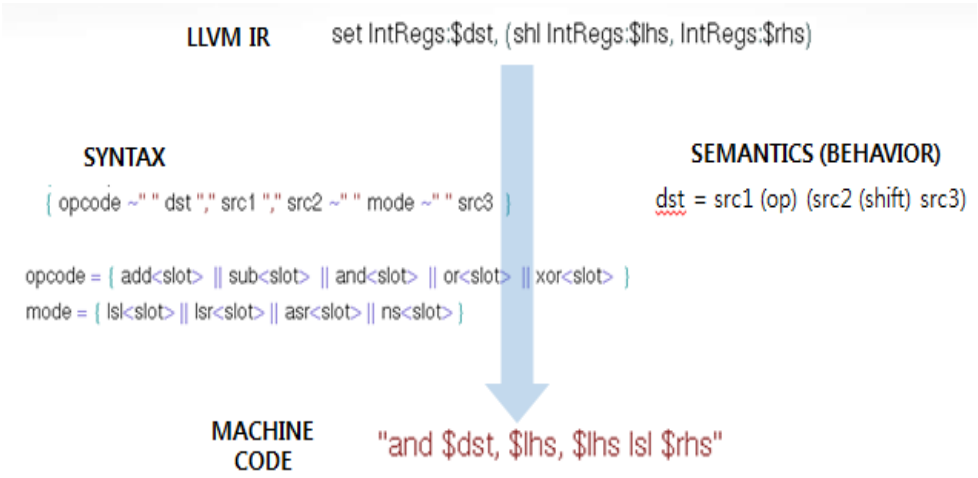

그림 11. LLVM IR to Machine Code Mapping

3.3.2 Frame Lowering

프로그램에서 함수 호출이 일어나면 해당 함수의 로컬 변수들과 함수의 인수를 저장하는 등 목적으로 사용할 공간이 메모리에 마련된다. 이를 해당 함수의 Frame이라 하는데, 함수가 호출될 때 Frame이 할당되고, 함수의 수행이 끝나고 원래 함수로 반환할 때에 Frame이 해제되어야 한다. 컴파일러는 target architecture의 적당한 명령어를 사용하여 이러한 일련의 과정들을 함수 처음과 끝 부분에 생성해 주어야 한다.

3.3.3 Expand Pseudo Instructions

LLVM IR에서 Machine code로의 mapping이 이루어 질 때, 모든 명령어가 실제 명령어로 생성되지는 않는다. 예를 들어, register에 큰 immediate 값을 할당하는 명령어는 LOAD_IMMEDIATE 라는 유사 코드(Pseudo code)로 생성이 되었다가 컴파일러의 마지막 단계에서 실제 명령어로 바뀌게 된다. 명령어에 encoding 될 수 있는 immediate의 크기는 제한되어 있기 때문에 그보다 큰 immediate을 register에 할당하기 위해서는 상위 bit와 하위 bit를 두 번에 걸쳐 할당하는 방식으로 해여 져야 한다. 이 작업은 mapping 단계에서 이루어지지 않고 LOAD_IMMEDIATE 명령어를 제일 마지막 단계에서 적절한 target architecture의 명령어로 바꿔주게 된다.

3.3.4 Assembly Printer

Processor Designer는 Assembler, Linker, Simulator/Debugger는 자동적으로 생성해 준다. LLVM Compiler에서는 C 코드를 target architecture의 Assembly code로만 바꾸어 주면 되는데, LISA로 기술된 Assembly syntax에 맞게 생성해주어야 한다. Assembly Printer는 LLVM Compiler의 가장 마지막 단계에서 완성된 코드를 target architecture의 assembly syntax에 맞게 출력해주는 단계이다. 이 부분을 저희 custom DSP의 syntax에 맞게 출력이 되도록 수정하였다.

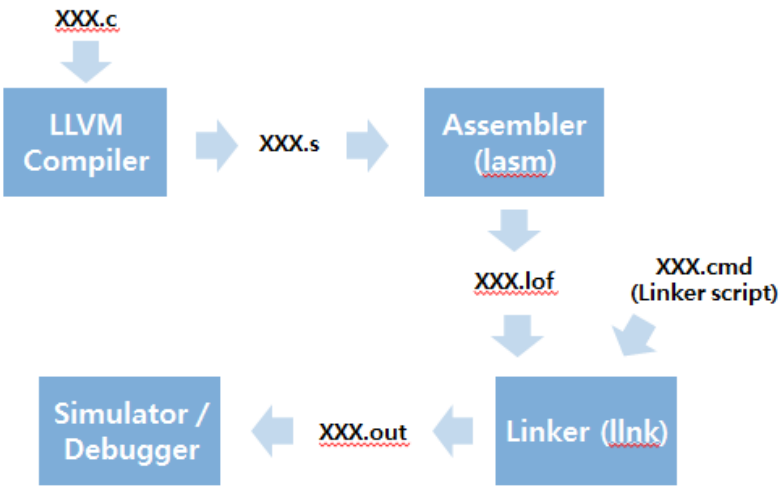

그림 12. Processor Designer tool chain

3.3.5 Instruction Scheduler

또한 VLIW는 따로 명령어 dependency를 파악하여 pipeline stall을 해주는 하드웨어가 없기 때문에, 컴파일러에서 명령어들 간의 latency 정보를 파악하여 스케쥴링을 해주어야 한다. Processor Designer에서 생성한 LLVM backend에는 기본 스케쥴러가 포함되어 있다. 여기에 수정이 필요한 부분은 각 명령어의 latency 정보를 추가해 주는 것이다. 그리고 컴파일러가 명령어들이 오직 지정된 slot에서 수행하도록 지지해줘야 한다.

3.3.6 Newlib 포팅

newlib은 임베디드 프로세서에서 c library를 쉽게 포팅하여 사용할 수 있도록 하는 library이다. newlib은 newlib과 libgloss로 나누어지는데 newlib은 stdlib.h, stdio.h 등을 포함하는 libc와 math.h 등을 포함하는 libm으로 이루어진다. libgloss는 어플리케이션 시작 코드나 system call을 지원하는 코드이다. newlib을 포팅하기 위해서 setjmp와 longjmp를 아키텍쳐에 맞게 포팅하였다. libgloss는 어플리케이션 시작 부분을 정의하는 crt0.S와 write 등 system call을 이용할 수 있도록 시뮬레이터와 연결시켜주는 rt_support.c를 구현하였다. 이 부분은 newlib을 test하는 부분에서 좀 더 자세히 설명하도록 한다.

3.3.7 특화 명령 생성 지원

DSP에서 추가된 특화 명령을 사용하기 위해서는 컴파일러에서 해당 명령어를 생성해 주어야 한다. 방법 중 하나는 LLVM IR 패턴을 분석해서 C code에서 해당 명령어가 사용될 만한 패턴을 찾아 자동으로 특화 명령을 생성해 주는 것이다. 그러나 보통 특화 명령어와 매칭되는 IR 패턴을 찾는 것이 쉽지 않아 효율적이지 않다. 다른 한가지 방법은 많은 DSP 제조업체에서 이용하는 것인데 바로 intrinsic function을 제공하여 C code에 function 형태로 해당 명령어를 사용하면 컴파일러가 이를 인식하여 해당 명령어를 생성해 주도록 한다. 프로그래머가 알고리즘에 대해 잘 알고 있기 때문에 그들은 어디에서 intrinsic function을 사용하면 될지 쉽게 파악할 수 있다.

Intrinsic function을 어플리케이션 특화 명령어로 번역하기 위해서는 컴파일러의 frontend를 수정하여야 컴파일러가 intrinsic function을 인식할 수 있게 해야 한다. 그렇지 않으면 컴파일러는 intrinsic function을 정의하지 않은 함수라고 인식하여 error가 생기게 된다. 수정한 후 컴파일러 frontend는 intrinsic function을 인식하여 그들을 intrinsic IR node로 번역하고 backend는 다른 일반 명령어와 같이 그들을 해당하는 명령어에 변환 시켜 준다.

프로그래머의 숙련도가 필요하지만, 임베디드 시스템의 성능을 최대로 끌어올리기 위해서는 효율적인 방법이다. 하여 우리의 custom DSP에서는 SIMD 연산과 특화 명령어에 해당하는 intrinsic function을 컴파일러에 추가하여, 해당 함수를 사용하면 해당 명령어가 생성되도록 하였다. 그림 13는 intrinsic function의 사용 예이다. int_로 시작하는 함수형태로 된 것이 intrinsic function이며, 해당 함수는 컴파일 과정에서 해당하는 특화 명령어로 1:1로 치환되게 된다.

```
over30 = int_bcmpgt(cdata4, 30);
over30_1 = int_bcmpgt(cdata4_1, 30);
uint8x4 tmp = int_bsub(cdata4, pdata4);
uint8x4 tmp_1 = int_bsub(cdata4_1, pdata4_1);
is_edge = int_bcmpgt(tmp, EDGEDEP) | int_bcmplt(tmp, -EDGEDEP);
is_edge_1 = int_bcmpgt(tmp_1, EDGEDEP) | int_bcmplt(tmp_1, -EDGEDEP);
```

그림 13. Intrinsic function 사용 예

3.3.8 기타 컴파일러 최적화

기타 컴파일러 최적화에는 Start up code 최적화: 실제 DSP에 올라가는 코드는 검증용 코드와는 달리 c library를 전혀 사용하지 않도록 하였고, 그에 따라 start up code도 main 함수를 호출하는 부분만으로 최소화하였다. Pre-RA scheduling: 컴파일러 과정 중 레지스터 할당 단계 이전에 스케쥴링을 한번 더 수행하도록 하였더니 성능이 더 좋아졌다. Callee saved register와 caller saved register 개수를 조정하여 성능을 더 높였다.

제4절 메모리 최적화

임베디드 시스템에서 성능 및 전력 소모량에 가장 영향력이 있는 구성요소 중의 하나가 메모리 시스템이다. 본 논문에서 제안한 DSP는 local SRAM을 많이 사용한다는 문제점이 있었다. 어플리케이션으로 사용한 손 인식 알고리즘을 돌린 결과 Instruction memory는 32KB, Data memory는 256KB를 필요로 하였다. 이는 임베디드 시스템에 있어 사이즈가 너무 크다.

이를 해결하기 위해 Cache를 이용할 수 있으나, VLIW의 특성 상 cache miss가 나면 모든 슬롯의 pipeline이 멈춰야 하여 penalty가 매우 크다. Cache miss는 동적인 특성이 있기 때문에 컴파일러가 이를 예측하고 스케쥴링에 반영하기가 매우 힘들다. 그 대신 Compute intensive 알고리즘의 특징 중 하나가 규칙적이다는 것인데, 이를 이용하여 software controlled memory 기법을 추가하였다. 손 인식 알고리즘의 경우에는 Image와 Finger Candidate을 저장하는 공간이 대부분을 차지하였다. 앞에서 설명했듯이 알고리즘에서 Image를 라인별로 처리하기에 Image는 전체를 SRAM에 저장해 놓는 대신 한 줄씩 읽어와서 처리하는 방식으로 바꿀 수 있다. Finger Candidate은 손을 여러 개까지 인식하는 경우를 대비해 공간을 많이 할당해 놓지만 대부분의 경우 할당된 메모리의 앞부분만 실제로 사용된다. 따라서 10개까지의 Finger Candidate만 메모리에 상시 할당을 해놓고 이를 초과하면 필요할 때마다 1개의 Finger Candidate을 DRAM에 저장하고 불러오는 방식으로 메모리 사용을 크게 줄일 수 있었다. (Instruction Memory와 Data memory 각각 16KB)

이를 위해 DMA에 SRAM과 DRAM간 데이터 전송을 요청하기 위한 ldstdram이란 특화 명령을 추가하였고, 해당 intrinsic function을 컴파일러에 추가하였다. 프로그래머는 그림 14와 같이 적절한 위치에 해당 intrinsic function을 추가해 SRAM과 DRAM간 데이터 전송을 할 주소, 그리고 그 데이터의 사이즈를 DSP에 전달함으로써 memory를 컨트롤 할 수 있다. 구체적인 동작은 아래와 같다. 프로그래머가 삽입한 intrinsic function이 컴파일을 통해 특화 명령어로 번역한다. DSP가 일단 ldstdram명령어를 수행하게 되면 먼저 DSP의 모든 동작을 멈추고,

output pin 중 하나인 REQ signal을 통해 DMA에 메모리 통신을 요구한다. DMA는 DSP로부터 SRAM과 DRAM간 데이터 전송을 할 주소, 그리고 그 데이터의 사이즈 등 정보를 받아 데이터 전송을 시작한다. 전송이 완료된 후 DMA는 input pin 중 하나인 ACK signal을 통해 DSP에 전송이 완료되었음을 알려 준다.

이 방법을 적용하여 우리는 Data memory의 사이즈를 원래의 256KB로부터 16KB로 줄일 수 있게 됐다.

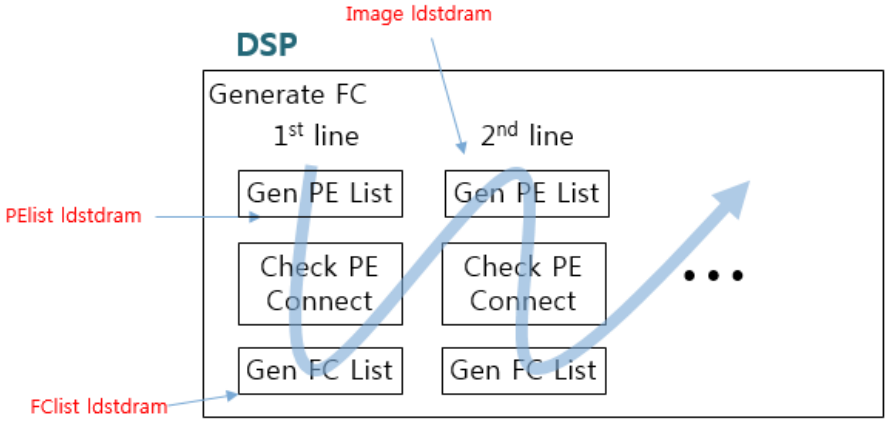

그림 14 메모리 컨트롤 예제

제 5 절 Test and verification

DSP의 정확성을 검증하기 위해서는 error가 생길 수 있는 어플리케이션, 컴파일러, 하드웨어 등을 모두 테스트해봐야 한다. 우리는 Processor Designer에서 제공해 주는 기능을 이용하여 여러 simulation 레벨에서 DSP system의 정확성을 확인하였다.

3.5.1 Sanity test

Processor designer에는 컴파일러의 검증을 위해 test code들과 이를 시험해 볼 수 있는 환경을 제공하고 있다. 그림 15는 그 시험 과정을 나타낸다. test code를 gcc로 컴파일하여 서버 환경에서 돌려서 그 결과와, 포팅한 llvm 컴파일러로 컴파일하여 시뮬레이터에서 돌린 결과를 비교하여 컴파일러를 검증하였다.

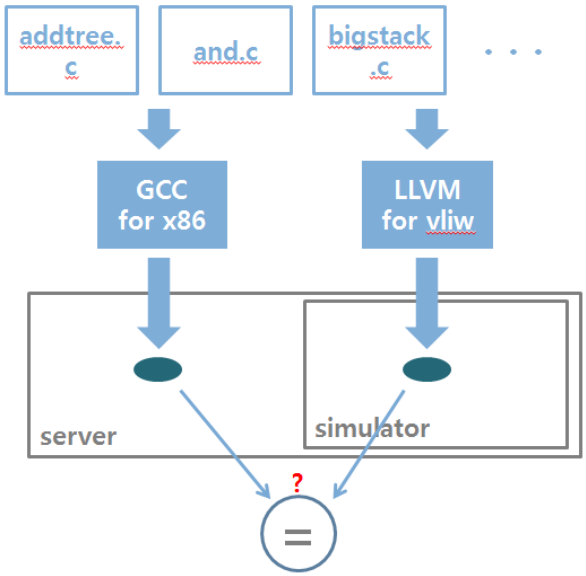

그림 15. Sanity test

3.5.2 Newlib test

Processor Designer에서는 포팅한 newlib이 제대로 돌아가는지도

검증할 수 있도록 newlib test용 test code들도 제공한다. Newlib은 임베디드 시스템에서 C standard library이다. 그 중 printf와 같은 경우 debugging을 위해서는 중요한 의미를 갖는다. 그림 16은 printf 함수가 어떻게 실행되는지를 나타낸다. printf 함수가 실행되면 연쇄적인 function call이 이루어 지다가 최종적으로 write이라는 함수 내부로 들어간다. 이 함수는 system call로서, OS에서 제공하는 함수이다. DSP에서는 OS가 돌아가는 것이 아니므로, 시뮬레이터에서 이 부분을 적절히 다루어야 한다. 따라서 libgloss에서 write 함수를 타겟 아키텍쳐에 맞게 변경해 주어야 한다. write에서는 sb uffer라는 내부 메모리 공간에 해당 system call을 수행하기 위해 정보들을 입력하고, syscall_handler를 호출한다. syscall_handler는 trap 명령으로 구성되어 있는데, 시뮬레이터에서 trap 명령을 마주치면, 그 후로는 시뮬레이터로 control이 넘어간다. 시뮬레이터에서는 manage_syscall이라는 함수가 불리우고, 여기서 sbuffer의 정보를 읽어서 syscall_write를 호출한다. syscall_write은 해당 OS의 함수 write을 호출하여 최종적으로 hello를 출력하게 된다.

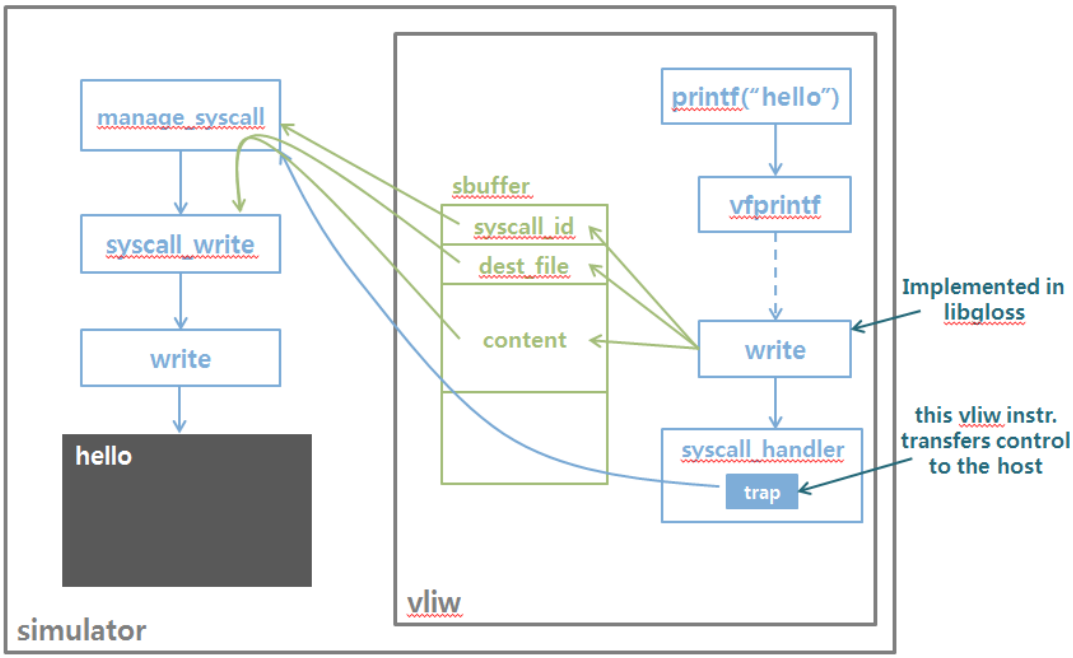

그림 16. printf 실행 과정

3.5.3 손 인식 알고리즘 검증

일반 명령어들이 프로세서와 컴파일러에서 제대로 동작하는지를 확인한 후 우리는 시뮬레이터를 통해 특화 명령어를 사용한 손 인식 알고리즘이 DSP 아키텍쳐에서 잘 돌아가는지 확인해 보았다. printf를 이용해 Finger Candidate가 제대로 계산되는지 확인하였고, 계산된 Finger Candidate가 정확한지 image 상에 띄워 보고 확인하였다. 손 인식 결과를 보기 위해서는 opencv library가 필요한데 이 부분은 DSP에서 돌아갈 수 있도록 컴파일 하지 않고 system call과 비슷한 방식으로 시뮬레이터에서 돌아갈 수 있도록 구현하였다.

3.5.4 RTL 검증

Simulator를 통한 검증이 끝난 후 Processor Designer의 RTL 생성 기능을 통해 DSP의 RTL을 생성하여 ModelSim을 통하여 LISA Simulator와 동일한 동작을 하는 지 검증하였다. ModelSim과 OpenCV를 연결하진 못하고, DSP 동작이 끝난 후 메모리에 올바른 값이 쓰였는지 확인하는 방식으로 검증하였다.

제 4 장 실험 결과

이 장에서는 우리의 손 인식 시스템의 prototype를 설명하였고 실험의 결과를 보여주었다. 제안된 시스템은 4개의 DSP를 DMA로 통해 통신하고, 매개 DSP 코어는 VLIW 아키텍쳐 형식을 가지고 있고 4-issue 32-bit width 아키텍쳐로 vision application의 Computation intensive task들에 특화되어 있다. 제안된 아키텍쳐의 성능을 비교 분석하기 위해 우리는 application을 전부 ARM Cortex-A9에서 돌린 것과 시스템에서 돌린 결과를 비교하였다. DSP 코어 성능은 4 슬롯에서 4개의 명령을 동시에 수행 가능하므로 4MIPS/MHz이다. 동작주파수는 FPGA에서 50MHz로 실험을 하였다. Gate count는 Global Foundry의 65nm 공정 라이브러리를 사용하여 Synopsys Design Compiler로 합성한 결과 memory를 제외한 부분은 DSP core 하나 당 72K 정도의 gate가 사용되었고 메모리를 포함한 경우 DSP core 하나 당 120K 정도의 gate가 사용된다. Cycle count는 ARM 프로세서 만으로 돌렸을 때보다 해당 부분이 3배 정도 빨라졌다.

제1절 실험 환경

Prototype으로 만든 손 인식 시스템 구성을 그림 17로 표현하였다. DMA와 통신을 할 수 있게 하기 위해서는 DSP와 어플리케이션을 수정할 필요가 있었다. DSP에 Status register를 추가하여 연산이 끝남을 표시하였다. 어플리케이션 코드의 입력 출력 데이터에 __attribute__ ((section(…))) 키워드를 어노테이션하여 그들의 위치를 정해 주었다. 다음 linker script를 사용하여 이런 변수를 메모리의 고정된 위치에 넣었다.

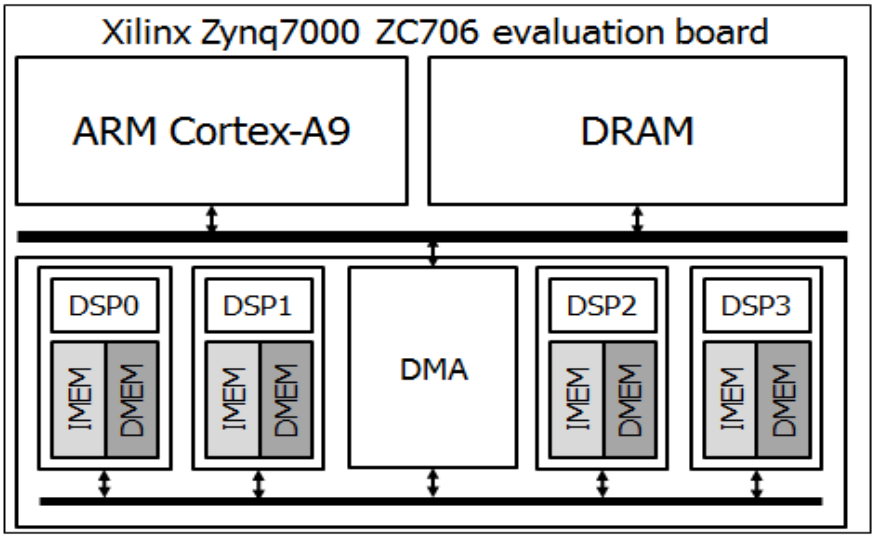

그림 17. Layout of the prototype system

제안된 시스템의 성능을 측정하기 위해 그림 18과 같은 실험환경을 구축하였다. 시스템의 동작 방식은 아래와 같다. Host PC에서 이미지를 보드에 보내 알고리즘을 시작한 후, 먼저 control 프로세서 (ARM) 가 이미지를 DRAM에서 4개의 DSP의 local instruction memory로 복사하도록 DMA에 명령을 내린다. 각 DSP는 똑 같은 image에 대해 0도, 90도, 180도 270도 방향으로 손 인식 알고리즘을 수행한다. DSP 연산이 끝나면 DSP의 output pin 중 하나인 done signal을 통해 DMA가 연산 완료를 확인하고, 결과 데이터를 DRAM에 복사한다. ARM 프로세서는 이 결과 데이터로부터 손의 위치 등을 계산한다. 결과는 USB를 통해 PC 화면으로 전송하여 볼 수 있다.

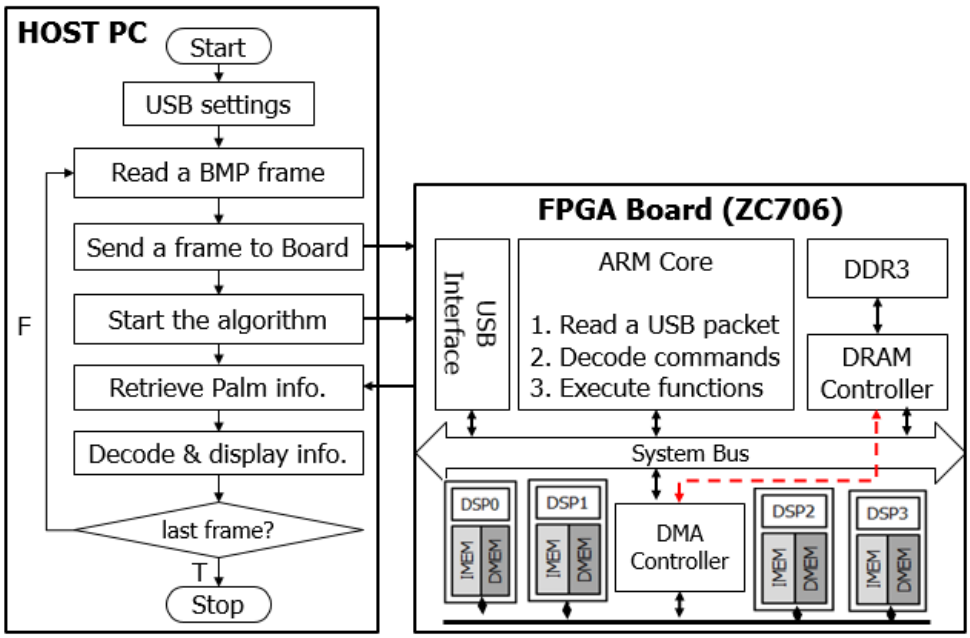
HOST PC
Start
USB settings
Read a BMP frame
Send a frame to Board
Start the algorithm
Retrieve Palm info.
Decode & display info.
last frame?
F
T
Stop
FPGA Board (ZC706)
USB Interface
ARM Core
1. Read a USB packet
2. Decode commands
3. Execute functions
DDR3
DRAM Controller
System Bus
DSP0
DSP1
DSP2
DSP3
IMEM
DMEM
DMA Controller

그림 18 실험환경

제2절 실험 결과

그림 19는 우리의 prototype 손 인식 시스템의 결과 이미지를 보인 것이고 표 1은 실험 결과를 보여준 표이다. DSP 코어 성능은 4 슬롯에서 4개의 명령을 동시에 수행 가능하므로 4MIPS/MHz이다. 동작주파수는 FPGA에서 50MHz로 실험을 하였고, Global Foundry의 65nm 공정 라이브러리로 합성하였을 때도 비슷한 수준의 결과가 나왔으나, 이는 메모리 사이즈가 컸을 때 해당 path가 critical path가 되어서 그랬고, 메모리 사이즈를 줄인 후의 결과는 좀더 나을 것이다.

주요 성능지표	단 위	결과	비고
1. DSP core 성능	MIPS/MHz	4	4 instruction/cycle
2. 동작주파수	MHz	50	
3. Frame rate	FPS	100@ 256x192	
4. Slot Utilization	%	35	
5. Gate Count	K	~120K per core	72K for logic ~48K for memory
6. Cycle Count		0.17M @ 50 MHz	Cortex-A9 9.8M @ 667 MHz

표 1. Experimental results

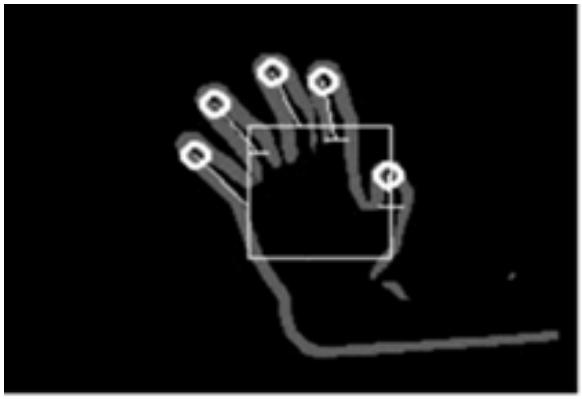

그림 19. Result of hand recognition

또한 Processor Designer에 의해 생성된 RTL은 자동 생성된 RTL이다 보니 hand optimize 과정을 거치면 최적화 될 여지가 남아있기 때문에 RTL를 최적화 한다면 동작 가능 주파수를 더 높일 수 있을 것이다.

Slot utilization은 상당히 낮은데, 그 이유는 computation intensive 한 부분만 DSP에서 돌린 것이 아니라, ARM 프로세서와의 load balancing을 위해 원래는 ARM에서 돌리려던 부분을 DSP에서 돌렸기 때문이다. 해당 부분은 if else문이 많아 VLIW 슬롯을 많이 활용하지 못하였다. 원래 DSP에서 돌리려던 성능에 가장 큰 영향을 미치는 부분은 55% 이상의 utilization을 보였다.

Gate count는 Global Foundry의 65nm 공정 라이브러리를 사용하여 Synopsys Design Compiler로 합성한 결과 memory를 제외한 부분은 VDSP core 하나 당 72K 정도의 gate가 사용되었다. SRAM memory cell 당 1.5 정도의 gate가 필요하다고 가정하면 메모리를 포함한 경우 VDSP core 하나 당 120K 정도의 gate가 사용된다.

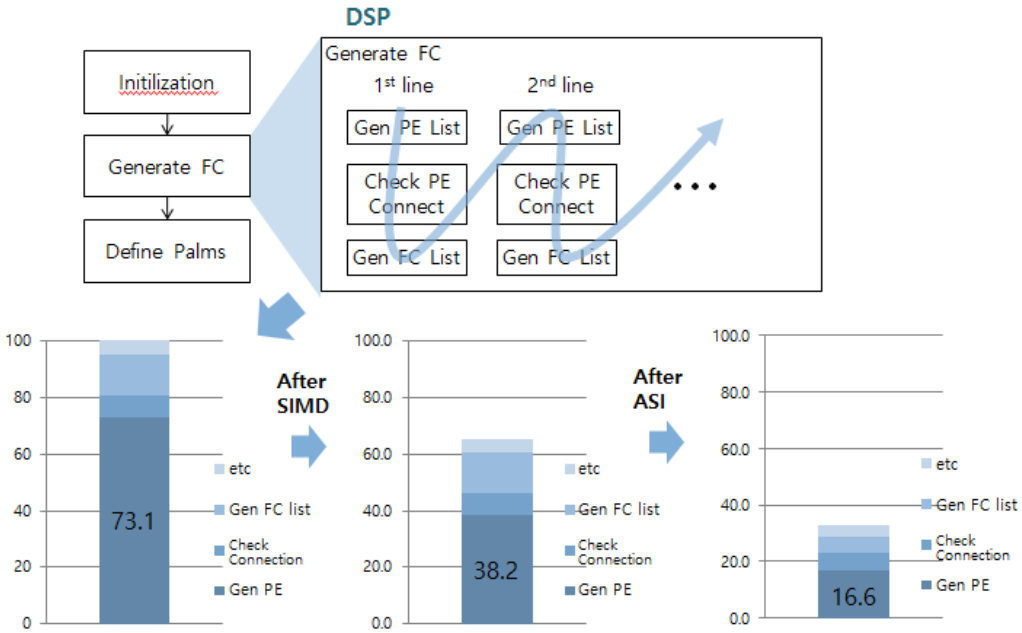

그림 20. 손 인식 실험 결과

표1을 보면, 알고리즘의 해당 부분을 돌리는데 소모한 시간은 ARM 프로세서 만으로 돌렸을 때보다 4배 정도 빨라졌다. Cycle count는 약 50배정도 적어 졌는데 이것은 특화 명령어를 사용해 가속화 된 원인도 있지만, DSP에서 실험한 알고리즘은 DSP에 맞게 최적화 하였으나 ARM에서는 그에 맞게 알고리즘을 최적화하지 않은 것도 있다. 그림 20은 손 인식 어플리케이션의 각 부분 별 성능 향상을 나타낸다. SIMD 명령과 ASI(Application Specific Instruction)을 사용하였을 때 해당 부분이 처음과 비교하여 3배정도 빨라진 것을 확인할 수 있다. 메모리를 최적화한 후 데이터 전송의 Cycle count 소모량은 원래의 0.1%도 차지하지 않아 별 차이가 없었다.

제 5 장 Conclusions

본 논문에서는 주로 비전 시스템을 위한 custom DSP을 제안하였다. DSP의 구현을 구체적으로 소개하였고 DSP를 위한 컴파일러의 포팅과 메모리의 최적화에 대해 설명하였다. 결과로 보아 ARM 프로세서만으로 돌렸을 때보다 약 4배 정도 빨랐고 또 그의 유연성을 보여 주었다. 그리고 software controlled memory기법을 사용함으로써 메모리 사용을 크게 줄이는데 큰 의미가 있었다.

제안된 DSP는 특화 명령어를 사용함에 있어 intrinsic function의 도움을 필요로 하여 다른 benchmark 측정 결과를 많이 보여주지 못하였다. 그러나 비전 영역 알고리즘들을 살펴보면 병렬성이 많고 반복 연산이 많은 application이 대부분이므로 우리가 제안한 ASIP은 유연성을 가지고 있으며 뚜렷한 개선점을 가져 주었다는 점에 큰 의미를 가지고 있다.

참고 문헌

[1] D. G. Lowe. Object recognition from local scale-invariant features. In Proceedings of International Conference on Computer Vision, 1999, pp.1150–1157.

[2] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. Proc. of CVPR'05, 2005, pp. 886–893.

[3] Synopsys Inc., Synopsys Processor Designer, http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx

[4] C. Lattner, V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," Proceedings of the 2004 International Symposium on Code Generation and Optimization, 2004, pp. 75-86.

[5] Xilinx. Inc. ZC-702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC, 2013. http://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik / ug850-zc702-eval-bd. pdf.

[6] S. Yu, R. Taib, and S. Lichman. "GestureCam: a smart camera for gesture recognition and gesture-controlled web navigation." In Control, Automation, Robotics and Vision, 2006. ICARCV'06. 9th International Conference on, pp. 1-6. IEEE, 2006.

[7] G. Amit, V. K. Sehrawat, and M. Khosla. "FPGA Based Real Time Human Hand Gesture Recognition System." Procedia Technology 6 (2012): 98-107.

[8] B. Mark, A. Couture-Beil, and G. Mori. "Real-time Motion-based Gesture Recognition Using the GPU." In MVA, pp. 9-12. 2009.

[9] J. Oh, G. Kim, I. Hong, J. Park, S. Lee, J. Kim, J. Woo, and H. Yoo. "Low-power, real-time object-recognition processors for mobile vision systems." Micro, IEEE 32, no. 6 (2012): 38-50.

[10] D. Patterson, J. Hennessy, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers, 2006.

[11] K. Shorin, T. Koga, L. Hanno, S. Nomoto, and S. Okazaki. "IMAPCAR2: A dynamic SIMD/MIMD mode switching processor for embedded systems." In Hot Chips 21: Symposium on High-Performance Chips. 2009.

[12] Mobileye, “EyeQ2,” 2008. [Online]. Available: http://www.mobileye.com/technology/processing-platforms/eyeq2/

[13] B. Brendan, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma. "Always-on Vision Processing Unit for Mobile Applications." IEEE Micro 2 (2015): 56-66

[14] Texas Instruments, TMS320C6000 Architecture Description, www.ti.com.

[15] P. Termont, L. De Strycker, J. Vandewege, J. Haitsma, A. Kalker, M. Maes, and G. Depovere, An implementation of a real-time digital watermarking process for broadcast monitoring on a trimedia VLIW Processor, Proc. IPA, 1999, pp. 775-779.

[16] P. Graham and B. Nelson. FPGA-based sonar processing. In Proceedings of the 1998 ACM/SIGDA eleventh international symposium on Field programmable gate arrays, Monterey, CA, Feb. 1998, pp 201– 208.

[17] D. Moloney, 1TOPS/W software programmable media processor, in HotChips HC23, Stanford, 2011.

Abstract

Wenquan Jin
Electrical and Computer Engineering
The Graduate School
Seoul National University

Computer vision applications are migrating to the mobile domain, but there are some difficulties mainly due to the computationally expansive nature of vision applications. On the other hand, there are various kinds of vision applications, which means using fixed hardware block will meet the performance criteria, but it will be possible to run only one application. To meet our needs, we can use another programmable component that is specialized for parallel processing and has domain specific instructions. In this paper, we introduce our work on developing a custom DSP for vision system that includes developing a custom processor with SPD tool, and porting LLVM compiler. As the target, we ported a hand recognition application to our system, which we have constructed on Xilinx Zynq evaluation board. Even though our DSP was run at much lower frequency, it could run compute intensive part of the application 3 times faster than ARM processor, greatly speeding up the whole appli-cation.

Keywords : Computer Vision; VLIW Architecture; LLVM Compiler; Digital Signal Processing; Recognition
Student Number :2014-22140