hw7 2017069598 박상지

2-1 ALU and data transfer instructions

#instruction set

컴퓨터가 제공하는 인터페이스로 이것을 이용해서 작업지시서를 만든다. 초창기에는 IS는 간단하게 구현하였다. 그러다가 메모리가 작아서 CISC가 유행을 했다. 메모리가 커지면서 RISC가 자리가 잡고, 후에 회사에서는 IS를 독점성을 위해 다르게 만들기 시작하였다. 하나의 프로세서가 있다고 가정할 때 사용자의 프로세서는 해당 응용프로그램에 맞게 돌아가게끔 만들어서 사용자는 그 응용프로그램만 사용하게끔 독점성을 이용한다. 하지만 이런 것들은 RISC기반으로 만들어진다.

#the MIPS instruction set

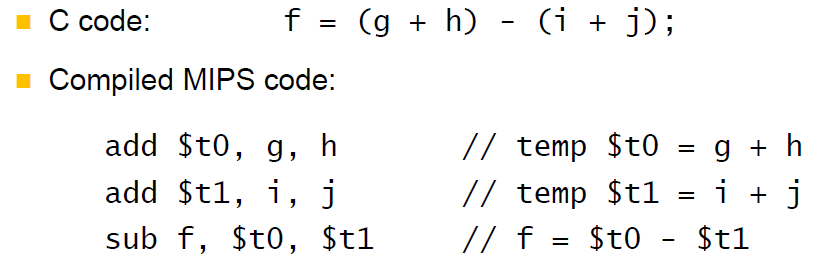
MIPS IS을 주되게 공부할 것이다. 임베디드 코어에 사용되기도 한다. 현대 ISA의 교육용으로 사용하기도 한다.

#ISA 감상: 생각의 초점

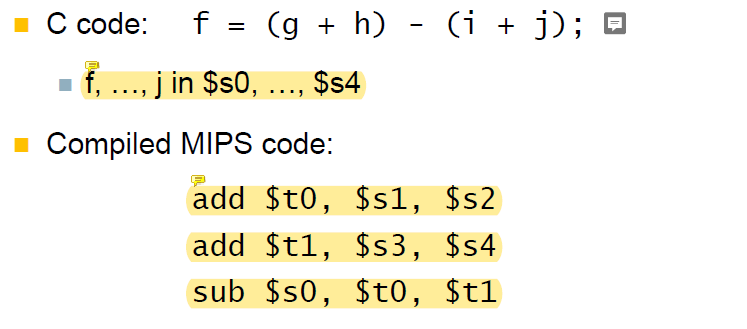
RISC ISA는 어떻게 생겼나? 왜 그렇게 생겼나?

RISC는 자주 나오는 operation을 single instruction을 만들어 준다. 자주 안 나오는 복잡한 것들은 여러 개의 instruction으로 만들어준다.

#Arithmetic Example



32개의 범용 레지스터를 갖고 있다. 32개는 상당히 많은 숫자여서 이것을 기억하기 편하게 어셈블러 언어를 사용한다. t는 임시 저장 값이고 s는 저장 값으로 사용한다. 이것은 레지스터 32개를 우리가 외우기 어렵기 때문이다.



f, g, … 는 s 레지스터에 저장했다고 가정하면 ALU instruction은 register-based mode 라고 불리는데 세 개의 operand를 사용한다. 뒤에 레지스터 두 개를 사용해서 해당 opcode에 따라 연산을 하고 앞에 있는 레지스터에 저장을 한다.

질문1)

만약에 f, g, … 해당 값들이 레지스터에 저장되어 있지 않고 메모리에 있다고 가정을 하면 해당 값들은 road를 하여서 레지스터에 각각 저장해줘야 한다.

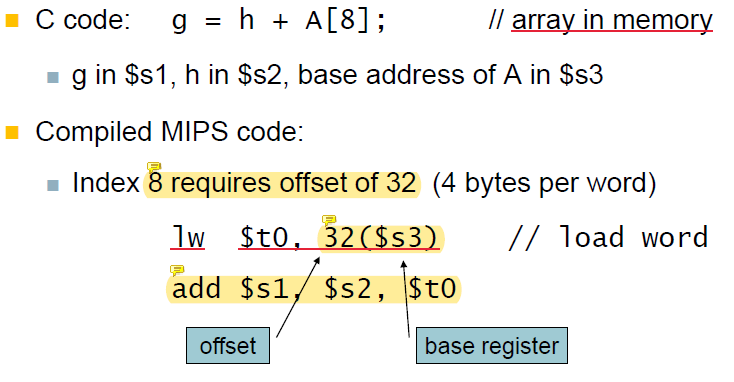
\*어느 value를 어떤 레지스터에 저장하는 것은 통상 컴파일러가 결정을 한다. 만약 어셈블리 언어라면 어셈블러 프로그래머가 직접 설정을 한다.

질문2) 범용 레지스터를 많이 만들면 좋지 않을까?

32개의 범용 레지스터를 쓰는 것은 몇 번 레지스터를 쓸려면 5bit을 사용하게 되는데, 레지스터를 몇 개를 만드는 것은 opcode, operand의 길이에 연관되어 있다. 그래서 64, 32, 16 레지스터 중에서 어느 레지스터를 사용해야 좋다. 라는 질문에서는 대답을 할 수 없다. 어떻게 설계를 하여 벤치 마킹을 빨리 돌리는게 목표이기 때문이다. 레지스터의 개수보다는 성능이 중요하다.

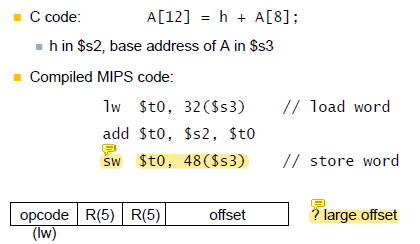
레지스터를 적게 사용하면 프로세서 입장에서는 레지스터에 있는 데이터를 접근하는 것이 빠르다. CPU가 메모리에 데이터를 접근하는 것은 수십억의 장소에서 하나를 찾는 것이다. 그래서 레지스터의 개수를 적게 써야 하나의 데이터를 찾고 읽고/쓰기 가 빨라진다. 즉 데이터 접근이 빨라진다.

#Memory Operand Example



A 배열을 메모리에서 읽어온다고 가정할 때, A 배열의 8번째의 값을 메모리에서 가져와야 한다. 메모리는 워드(word) 단위로 떨어져 있기 때문에 해당 배열은 8word, 즉 32(byte)의 상수 값을 갖게 된다. memory operand는 2개의 instruction이 있고 load 와 store이 존재한다. 레지스터 2개와 상수 값을 갖게 되고 이것을 base addressing mode 또는 offset addressing mode라고 한다. 뒤에 있는 레지스터를 읽어서 상수 값을 더해서 읽어서 그 데이터를 앞에 있는 레지스터에 저장한다.

32($s3) 은 기준이 되는 레지스터 s3에서 얼만큼 떨어져 있는지 32byte(8word)



store에서는 앞에 있는 레지스터의 데이터를 s3 레지스터에 48byte(12word)에 있는 레지스터 저장한다.

질문) load instruction은 뒤에 있는 레지스터에 왜 기준 레지스터와 상수 값을 주어지냐?

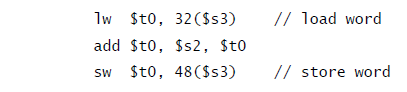
만약 32-bit address를 주게 되면 이미 instruction 자체가 32bit를 넘게 사용하기 때문에 공간을 두 개를 써야 된다. 그렇게 되면 접근을 두 번 해야 하기 때문에 성능이 떨어진다.

#Register Allocation and Spill

메모리에서 데이터를 road를 하면 레지스터에 저장된다. 어느 레지스터, 즉 몇 번 레지스터에 저장하는 것을 결정하는 게 컴파일러이다. 이런 작업을 register allocation이라 한다.

메모리에서 데이터를 레지스터 32개에 road를 하는데 만약 32개의 레지스터를 전부 다 사용하였다면 다음 메모리에서 읽은 데이터를 레지스터에 저장해야 하는데 사용하지 않은 레지스터, 즉 어느 레지스터에 저장해야 하는지 이런 작업을 register spill이라 한다. 이것도 역시 컴파일러에서 처리한다.

질문) Load-store Architecture 퀴즈



1. 몇 번의 메모리 접근을 하는가?

load instruction을 실행하려면 fetch-decode-execute가 일어나면 이 과정에서 메모리 접근이 3번이 일어난다. 그리고 해당 load, store instruction을 실행시킬 때 execute가 한번씩 일어나기 때문에 총 5번이 일어난다.

1. 몇 번의 instruction 접근을 하는가?

3번

1. 몇 번의 data 접근을 하는가?

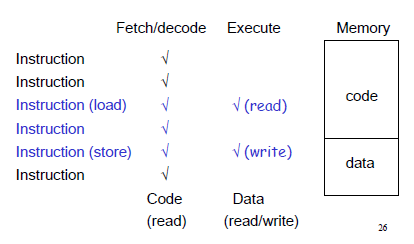
2번

1. 몇 번의 메모리 read를 하는가?

load instruction에서 fetch-decode-execute에서 메모리 read가 3번 일어나며 마지막 load instruction의 execute 자체가 read이기 때문에 총 메모리 read는 4번 일어난다.

1. 몇 번의 메모리 write를 하는가?

store의 execute가 write이기 때문에 1번 일어난다.



#the Constant Zero

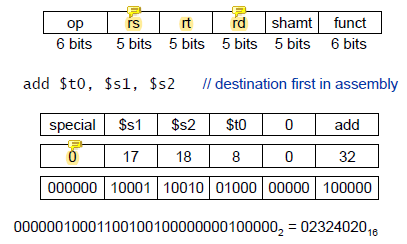
0번째 레지스터의 값을 항상 상수 값 0이 저장되어 있다. 그리고 데이터의 값은 변하지 않는다. 그렇다면 32개 중에서 하나를 사용하지 않는 꼴인데 왜 이렇게 사용하는가?

* 다른 instruction을 편리하게 사용하기 때문이다.
* 레지스터 사이에서 move를 생각해 보면 따로 move라는 instruction을 만들 필요 없이 하나의 레지스터와 zero 레지스터를 add를 하면 다른 레지스터에 하나의 레지스터의 값이 저장하게 되어진다.
* 레지스터의 값을 없앨 경우, 즉 clear를 할 때에는 해당 레지스터에서 zero 레지스터 두 개를 더해주면 해당 레지스터는 0의 값을 갖게 된다.

#Stored Program Computers

데이터를 binary로 표현하는 것처럼 프로그램 instruction도 binary로 표현되어야 한다.

#R-format Example



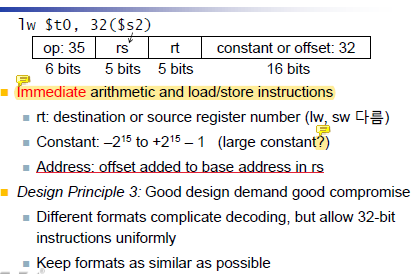
R-format은 그림과 같이 op, register source 두 개와 마지막 저장해주는 register destination 그리고 shift 연산을 할 때 사용되는 Shamt와 마지막 function 자리가 존재한다. funct에서는 opcode의 모자란 부분을 채워준다.

레지스터 기반으로 돌아가는 ALU 연산의 op는 0이다. 모든 opcode을 구분하기 위해서 function 코드로 구분한다. special에서 0을 보고 ALU 연산을 사용했다는 것을 짐작할 수 있다.

질문) 특정한 funct를 만들지 말고 opcode를 8-bit으로 왜 사용하지 않는가?

다른 field의 공간이 제한되어진다. opcode를 6-bit으로 제한 이유는 나머지 다른 field를 더 잘 사용하기 위함이다.

#I-format instructions



I-format에서는 op, register source, register destination, offset 필드가 존재한다. 상수 값은 instruction안에 들어 있기 때문에 레지스터에 따로 갈 필요 없이 instruction 안에 32의 값을 즉시 알 수 있다. 이것을 immediate number이라 한다.

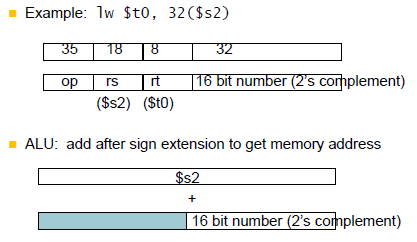
#To think about: R and I formats

format을 두 개가 있는 게 무슨 의미가 있을까? 다른 format을 처리하는 하드웨어를 만들어 줘야 한다. 많은 format을 만들면 하드웨어는 해당하는 format을 만들어 줘야한다. 그렇게 되면 하드웨어는 복잡해진다. MIPS는 format이 3개 밖에 없어서 하드웨어가 단순하고 빠르게 만들어 줄 수 있다.

opcode을 몇 bit를 사용할 것인가 그리고 레지스터를 몇 개를 사용할 것인지 대한 해당사항들은 16-bit 상수 값을 설정한 것에 연관되어 있다.

물론 6-bit opcode, 16 register, 18-bit offset, 8-bit opcode, 32 register, 14 bit offset 등 후보 format을 생각할 수 있다. 이 후보들 중에서 벤치마킹을 돌려서 가장 성능이 좋은 것을 선택을 하는 것이다.

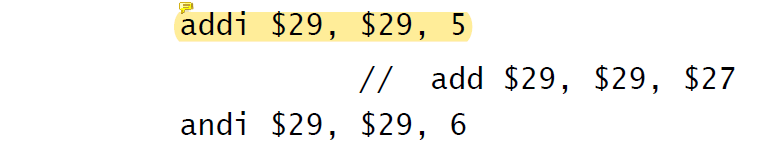
#Sign Extension



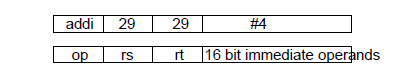
sigh bit을 16개를 그대로 복제하면 값을 유지하면서 32-bit을 만들어 줄 수가 있다. 그 다음 ALU에 더해달라 하면 되는 것이다. 이것을 Sign Extension이라 한다.

#Revisit ALU instructions

대부분 프로그래머의 코드의 25%은 ALU 연산을 사용을 한다. 그 중에서 A = A+ 5 와 같은 연산을 사용한다. 이 통계는 역시 벤치 마킹에서 프로그램을 돌리 때의 통계이다. R-format에서 레지스터 기반의 ALU 연산을 배웠지만 지금은 상수 값을 따로 처리해야한다. 통계처럼 이렇게 자주 나오는 Instruction을 single instruction으로는 표현되지 않는다. 상수 값을 메모리에서 읽어야 하기 때문에 이런 자주 나오는 instruction을 따로 만들어주는데



그림과 같이 add immediate instruction을 새로 만들어 준다. 29번 레지스터 값과 상수 5 값을 더해서 해당 29번 레지스터에 값을 넣어준다. 상수 5는 instruction에 존재하기 때문에 더 이상 메모리에서 찾을 필요가 없어진다. 이렇게 instruction 안에 상수 값이 존재하는 것을 immediate라고 부른다. 즉 immediate addressing mode라 한다. add i에서는 레지스터 두 개와 immediate number가 나온다.



상수(offset)을 사용하기 때문에 load 와 operand 모양은 비슷하여 addi 에서는 I-format을 형태를 사용한다. 상수는 연산을 할 때 2의 보수로 들어간다. 연산을 하기 전에 32-bit으로 만들어 주고 sign extension을 한다.

질문) 16-bit immediate를 사용하는 것은 좋은 선택인가?

이것도 역시 벤치마킹을 분석하여서 결정을 한다. 16-bit 정도 할당하면 single instruction이 되면 16-bit을 사용하고 경우에 따라 다른 결정을 할 수 있다. 결론은 벤치 마킹을 통해서 성능을 분석하여 좋은 성능을 고르면 되는 것이다.

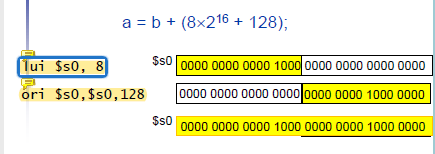
#Large Constants



질문) add i에서 16-bit immediate를 사용한다. 그렇다면 immediate operands에서 16-bit 보다 더 큰 상수 값을 요구하면 어떻게 처리해야 하는가?

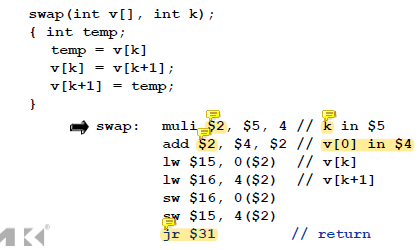
add i로 사용 불가하다. 큰 수를 레지스터에 저장을 해야 한다. machine instruction이 두개가 필요하며, 그러므로 큰 수가 나올 경우 instruction을 세 개를 사용해야 한다. 하지만 이런 경우는 드문 일은 아니다.

#32- bit Constants



상수 값이 16-bit를 넘어가면 instruction을 2개를 써야 한다고 했다. lui, ori 연산을 사용해야 하는데, lui(load upon immediate)는 8이라는 수는 32-bit에서 상단 부분 16-bit에서 8의 값으로 저장된다. 왜냐하면 해당 그림에서 8 x 2^16은 16비트 이상의 8의 값이기 때문이다. ori(or immediate)는 하단 16비트에서 128의 값을 저장한다.

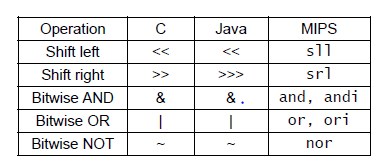
#example



해당 swap 코드를 MIPS 코드로 작성을 하면, 처음에 인덱스 K 부분을 처리해야 하는데 주소는 1워드(word) 만큼 떨어져 있기 때문에 4바이트(byte)을 5번 레지스터(K해당 레지스터)에 곱해서 2번 레지스터 저장을 한다. K인덱스와 V에 해당한 레지스터를 더해주면 V의 배열 값이 생성되고 Load를 해서 V[K] 값과 V[K+1]을 각 레지스터 15, 16번에 할당해 준다. Store 해줄 때에는 반대로 저장을 해주면 swap이 끝난다. 마지막 jr instruction을 return에 해당한 부분이고 후에 정리할 계획이다.

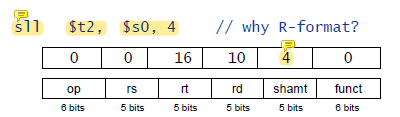
#Logical Operations

논리 연산은 32-bit word를 통째로 연산 하는게 아니라 32-bit의 일부 bit을 연산하는 것이다.



#Shift Operations

shift 연산은 shift left/right logical 2개로 구성되어 있다. 왼쪽으로 shift를 할 때에는 옆으로 옮길 때마다 0-bit로 채워진다. 그래서 sll(shift left logical)에서는 곱하기 개념이 담겨있다.



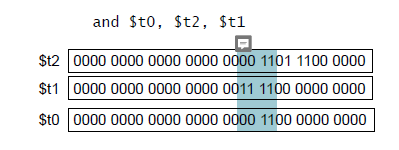
레지스터 두 개와 상수 값이 사용되며 레지스터 s0에 4-bit 만큼 shift하여 t2에 저장한다. 이것은 t2의 값에 s0 레지스터의 값을 2^4 = 16을 곱한 것과 동일하다.

질문) 왜 R-format을 사용하였는가?

shift는 최대 32-bit이다. shamt field는 5-bit으로 표현할 수 있으면, R-format으로 사용하여도 무관하다. I-format을 사용하지 않은 이유는 R-format에서는 opcode와 funct가 존재하는데 프로세서는 특정 field에 따라 opcode를 설정한다. 하지만 I-format에서는 opcode에 대한 여유가 없기 때문에 shift 연산을 할 때에는 R-format을 사용한다.

shift right logic을 할 때에는 반대로 오른쪽으로 shift되며 0-bit로 채워진다. 그래서 나누기 개념이 담겨있다.

#Bitwise AND operations

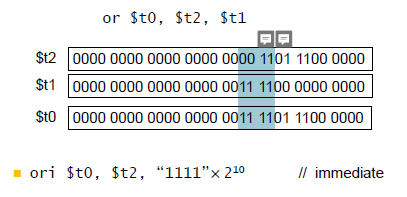


bitwise AND 연산은 32-bit의 일부 비트만 사용하여 연산하는 논리적 operation이다. 색깔이 입힌 부분을 제외한 나머지 부분은 0으로 clear하고 어떤 비트를 남기고, 나머지는 제로로 만들 때 사용한다. 즉 32-bit에서 내가 원하는 부분을 만들 때 사용하는 연산이다.



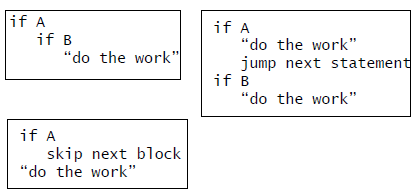
상수는 16-bit의 immediate number가 주어진다. t1 레지스터 랑 같은 값 1111에 10칸을 shift하였다. 논리적 연산이기 때문에 2의 보수가 아니다. 그래서 나머지 빈 공간은 0으로 채워 준다.

#Bitwise OR operations



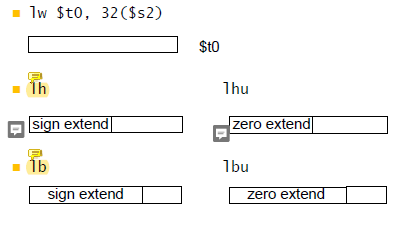
내가 원하는 부분을 1로 설정하고 싶을 때 사용하며, 나머지 부분은 신경 쓰지 않는다. 임베디드 시스템에서 스위치와 램프를 컨트롤 할 때 bit 0은 스위치 상태를 나타내고 bit 1은 램프의 on/off 상태를 나타내면 32-bit의 내가 원하는 부분을 바꿔서 스위치, 램프를 컨트롤한다.

#To think about



프로그램을 짤 때 and, or, not statement를 사용한다. 사실 and operate는 첫번째 그림처럼 사용할 수 있지만 프로그래머들은 선호하지 않기 때문에 && statement를 사용한다. 꼭 필요하지 않지만 프로그래머의 편리함을 위해 만들어 놓은 것을 syntactic sugar이라 한다.

#Byte/Halfword Operations

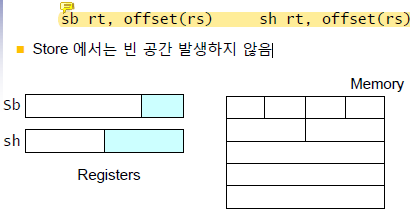


load 연산만 해도 5개의 instruction이 나온다. lh, lhu, lb, lbu, lw 이렇게 5개의 instruction이 나오는데 lh와 lb는 load half word와 load byte로 크기를 유지하면서 32-bit로 늘려줘야 한다. 즉 sign extends를 해줘야 하는데, 지정된 주소에 byte/halfword로 읽고 sign extends를 해준다. 그런데 2의 보수가 아닌 char, unsigned number을 갖고 올 때에는 나머지 공간을 zero로 채워 줘야한다.

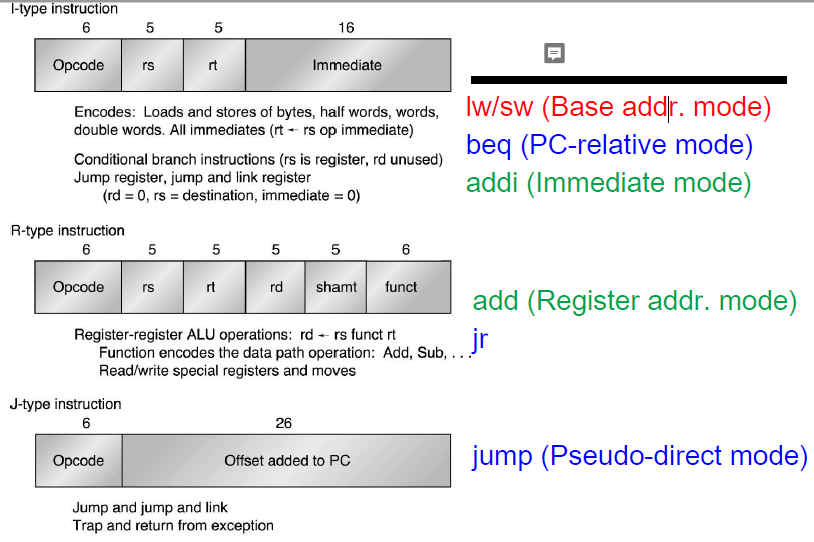
이 차이를 구분하기 위해서 operation을 두개를 각각 만들어 준거다. load만 해도 5개의 instruction이 나온다. 그래서 opcode가 부족한 현상이 일어난 것이다.

질문) bit operation, shift을 배웠으니 byte를 가지고 올 때도 이런 논리적 연산으로 만들면 되지 않을까?

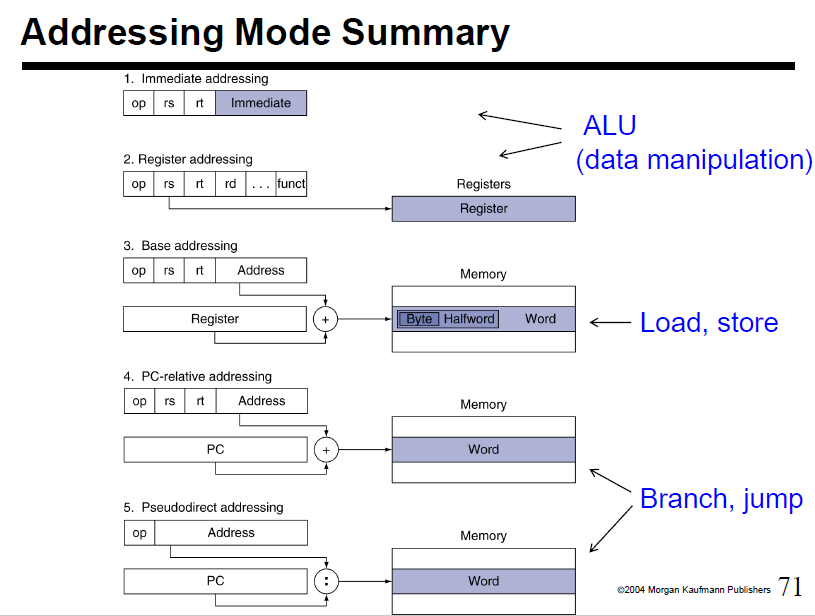
instruction이 추가로 드는데, byte/halfword와 같은 것들은 자주 쓰이는 것이기 때문에 새로운 instruction을 만들어 주는 게 났다.



store 연산에서는 두 개의 instruction이 추가되는데 지정된 주소에 집어넣으면 되기 때문에 빈 공간이 발생하지 않는다. 주변의 데이터를 건들면 큰일난다. (crash(충돌)이 일어나기 때문이다.)



MIPS에는 instruction이 전부 32-bit인데 이것을 어떻게 잘라 쓸 것인지에 대한 format을 배웠다. I-type은 load, store, addi에서 사용되고 base addressing mode, immediate mode에 해당된다. R-format에서는 ALU 연산을 사용할 때 또는 shift, logical 연산을 사용할 때 사용되며 register addressing mode에 해당된다. 후에 J-type을 배운다.



질문) RISC ISA는 어떻게 생겼나? 왜 그렇게 생겼나?

벤치 마킹 프로그램이 중요하다. 프로세서 기계는 소프트웨어를 빠르게 돌리는 것이다. ISA는 하드웨어 소프트웨어 instruction을 다루는 것이면, RISC은 자주 나오는 operation을 single operation으로 표현하는 게 목표이고 자주 나오지 않는 operation은 여러 개의 operation으로 표현되어도 괜찮다.

#64-bit MIPS ISA

우리는 64-bit 프로세서에서 살고 있는데 왜 32-bit MIPS ISA를 배우고 있는가??

추가된 32-bit를 우리가 어떻게 사용해서 성능을 내는가?

결론은 MIPS 64-bit instruction set을 찾아보면 우리가 본 위에 그림이 나온다. 추가적인 32-bit을 사용하지 않는다. 레지스터 숫자도 늘리지 않는다. 똑 같은 32-bit instruction을 사용한다. 왜 그럴까?

1. MIPS 64는 MIPS 32에서 진화된 IS이다. MIPS 64을 많은 사람들을 쓰게 하기위해서 필요한 것만 추가해고 나머지들은 compatible해야 한다. 우리가 배운 것은 MIPS 32에서 핵심적인 내용이며, 이것 역시 MIPS 64에서도 핵심적인 내용이다.
2. 메모리 접근을 할 때 instruction 길이는 32-bit이기 때문에 두 개의 instruction을 한꺼번에 가져올 수 있는 것이다. instruction을 빠르게 가져오는 게 가장 중요하기 때문에 이 부분에서 굉장한 speed-up을 제공한다.