2장

명령어 집합 -> 다른컴퓨터는 다른명령어 set을가짐(매우 유사한점도 많음), early computer은 매우 단순한 명령어집합 가짐. 요즘 컴퓨터도 단순함.

내장 프로그램(stored-program concept)

-> 여러종류의 데이터와 명령어를 메모리에 숫자로 저장할 수 있다는 개념

MIPS 명령어 집합

add a, b, c # a gets b + c -> 만약 변수가 4개이면 명령어 집합이 3개 필요

add a, a, d

add a, a, e => 네개의 변수 합 구하기

디자인 설계 원칙1 -> 간단하게 하기 위해서는 규칙적인 것이 좋다.

- Regularity는 실행을 단순화 한다, 단순함은 적은가격에 높은 성능을 낼 수 있다.

Register Operands -> 레지스터는 하드웨어 설계의 기본요소이다. MIPS 구조에서는 레지스터의 크기는 32비트이다. 이것을 워드라고 한다. $t0~t9는 임시 변수, $s0~s7은 저장변수

디자인 설계 원칙2. 작은것이 빠르다 -> 레지스터 개수를 32개로 제한하는 이유

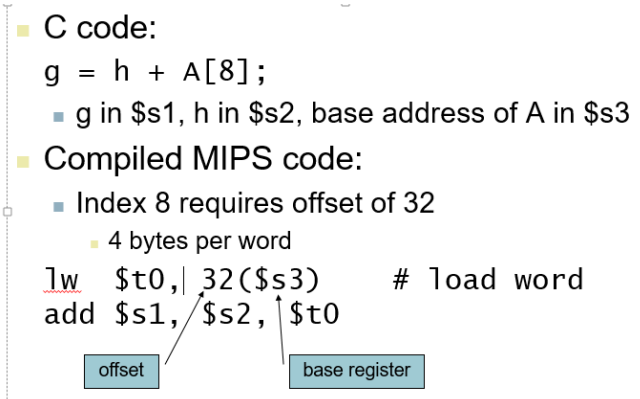
Memory Operands

-> 메인메모리는 복합데이터를 사용

-> 배열이나 구조체는 메모리에 저장

-> Load : 메모리에서 레지스터, Store : 레지스터에서 메로리로

-> 메모리는 byte(8bit)



-> 메모리내에서 데이터는 자연스러운 경계를 지켜서 정렬되어야 한다. 연속된 워드의 주소는 4의 배수여야 한다.(1 word= 4 byte=32 bit) -> 정렬제약(alignment restriction)

-> MIPS는 Big Endian(제일 왼쪽, 즉 최상위 바이트 주소를 워드 주소로 사용함)

-> sw $s1, 24($s2) : memory[$s2+24]=$s1 (store) (GiB : 2^30, TiB: 2^40)

Register VS Memory

Registers가 메모리보다 접근이 빠르다 -> **디자인 설계원칙 "작을수록 빠르다"**

메모리 데이터에서 작동하려면 로드 및 저장이 필요

컴파일러는 변수에 대해 가능한 한 레지스터를 사용해야 한다. -> 사용빈도가 낮은 변수를 메모리에 넣는다(spill)

2.4-2.5

MSB: 가장 왼쪽의 비트 31**, LSB:** 가장 오른쪽의 비트 0

Immediate Operands(수치 피연산자)

addi $s3, $s3, 4 -> 이것처럼 상수를 더하는데 사용하는 addi을 수치 피연산자라고 한다.

빼기 수치 피연산자는 없다. addi operand을 이용하자. 음의 상수도 지원가능하기 때문

상수 0은 유용한 여러 변형을 제공해 단순한 명령어 집합을 가능. 예를 들어 move(복사)는 피연사중 하나가 0인 add명령어 이다.(add $t2, $s1, $zero) MIPS에서는 레지스터 $zero을 값 0으로 묶어 두도록 회로가 구성.

디자인 설계원칙 3. 자주 생기는 일을 빠르게 하라

->작은 상수는 일반적이다.

->Immediate operand은 load 명령어를 피하게 해준다.

칩내의 레지스터 개수 -> 매우느리게 증가한다

-> 이유 : 프로그램은 보통 컴퓨터 언어의 형태로 배포되므로 명령어 집합 구조에는 일종의 관성이 있다. 따라서 레지스터 개수는 새로운 명령어 집합이 실용화되는 만큼 느리다.

Unsigned Binary Integers

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

->Range: 0 to $+2^n - 1$

sign and magnitude(부호와 크기) -> 한비트를 부호비트로 두어서 계산하는 것, 하지만 양과음의 0이 존재하게 된다. 그래서 안씀, 덧셈,뺄셈시에 부호비트를 안건드림

2s-Complement Signed Integers

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

->Range: -2^{n-1} to $+2^{n-1} - 1$

비트 31이 부호비트이다. -> 1이면 음수 ,0이면 양수, MSB은 2^(-31)이다.

⇨ $-(-2^{n-1})$ can't be represented

- ⇨ 0: 0000 0000 … 0000
- ⇨ –1: 1111 1111 … 1111
- ⇨ Most-negative: 1000 0000 … 0000
- ⇨ Most-positive: 0111 1111 … 1111

Signed Negation(2의보수)

두가지 연산법

1. complemet하고 1을 더한다.) -> 양수에서 음수구하기

2. Sign Extension(부호확장)

Sign Extension(부호확장)

Load, store, branch, add 그리고 set on less than 명령어의 수치 필드에는 2의보수 16비트 이진수가 들어가므로 32비트 레지스터에 더하려면 컴퓨터는 16비트->32비트로 바꿔야 한다.

Addi lb, lh, beq, bnne extend 한것들

1. sign bit(부호비트)를 왼쪽 부분에 전부 채운다.

2. 원래의 값은 그대로 오른쪽부분에 복사한다.

1의보수법은 0이 두개표현됨 맨 앞비트가 부호비트이고 가장 큰음수는 10000, 큰 양수는 011111 이런식으로 표현한다. 양수와 음수 개수가 같음. 1의 보수 덧셈기는 맨끝의 올림수를 처리하기 위해 한단계를 더 필요하다. (x의보수는 2^n-x-1)

Representing Instructions

명령어도 2진수로 표현된다.(기계어) , **외우기!** , 모든 mips 명령어는 예외없이 32비트이다.

$t0 – $t7 are reg's 8 – 15

$t8 – $t9 are reg's 24 – 25

$s0 – $s7 are reg's 16 – 23

$a0 – $a3: arguments (reg's 4 – 7)

$v0, $v1: result values (reg's 2 and 3)

MIPS R-format Instructions

OP	rs	rt	rd	shamt	funct
6bit	5bit	5bit	5bit	5bit	6bit

op: operation code (opcode), 연산의 종류

rs: first source register number

rt: second source register number

rd: destination register number

shamt: shift amount (00000 for now)

funct: function code (extends opcode), 연산의 종류에서 그 중에 한 연산을 구체적 지정

Hexadecimal

4비트씩 나눠서 2진수->16진수, 16진수->2진수변환하면 된다.

MIPS I-format Instructions

위에 R-format보다 필드 길이(5bit)가 길어야 하면 문제가 생긴다. 이런 문제 때문에 모든 명령어의 길이를 같게 하고 싶은 생각과 명령어 형식을 한가지로 통일하고 싶은 생각 사이에 충돌

디자인설계원칙4. 좋은 설계에는 적당한 절충이 필요하다.

->다양한 형식은 디코딩을 복잡하게하지만 32 비트 명령을 균일하게 허용합니다.

->형식을 가능한 한 유사하게 유지하십시오

MIPS 설계자들이 택한 절충안은 모든 명령어의 길이를 같게 하되, 명령어에 따른 형식을 다르게. => I-fromat

OP	rs	rt	constant or address
6 bits	5bits	5bits	16bits

-> lw나 sw에서 rt는 목적지 레지스터 번호, rs는 베이스 레지스터 번호

16비트 주소를 사용하므로 lw 명령은 베이스 레지스터 rs에 저장된 주소를 기준으로 –2^15 to +2^15 – 1를 지정가능, 마찬가지로 addi에서 사용할 수 있는 상수도 임.

요점정리

오늘날의 컴퓨터는 두가지 중요한 원리 바탕

->명령어는 숫자로 표현, 명령어와 데이터는 메모리에 저장

->프로그램은 메모리에 기억되어 있어서 숫자처럼 읽고 쓸 수 있다

이것이 내장 프로그램의 개념이다.(stored-program concept)

프로그램은 다른프로그램을 작동시킬수 있다. Ex) compiler, linker

이진 호환성(Binary compatibility)으로 컴파일 된 프로그램이 다른 컴퓨터에서 작동 할 수 있습니다.

2.6 Logical Operations

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

논리연산 명령어는 한 워드의 비트 그룹을 추출하고 삽입하는 데 유용합니다.

Shift Operations

R-format에서 shamt는 자리이동량이다.

sll-> 왼쪽으로 자리이동하고 이동후 빈자리는 0으로 채운다,

i bit만큼 이동하면 2^i배한것이다. 밑에 것은 그만큼 나뉘어진것이다.

srl-> 오른쪽으로 자리이동하고 이동 후 빈자리는 0으로 채운다.

sll은 OP코드와 funct 필드가 0, **rs 필드는 사용x**

AND Operations

두비트가 1일경우만 1이나온다. Mask-> 어떤 비트 패턴에서 0의 위치에 해당하는 비트들을 강제로 0으로 만드는데 사용

OR Operations -> Useful to include bits in a word

NOT Operations -> 피연산자 하나를 받아서 1->0, 0->1으로 invert

NOR Operations -> 피연산자 3개 필요, a NOR b == NOT (a OR b), 상수 피연산자 구현x

nor $t0, $t1, $zero($zero 항상 0을 읽음, 이거는 t1값을 0,1 invert해서 t0에 저장하는 것임.)

andi, ori -> andi와 ori는 상위 16비트에 0을 삽입하여 32비트 정수를 만들 수 있다. 이는 addi랑 다르다. addi는 부호비트를 확장시킨다.

Conditional OperationsR

beq rs, rt, L1 ->if (rs == rt) branch to instruction labeled L1;

bne rs, rt, L1 ->if (rs != rt) branch to instruction labeled L1;

j L1 -> L1로 무조건 분기하라

Assembler calculates addresses8/

Compiling Loop Statements -> p97

Basic Block

기본블록(basic block)은 분기 명령을 포함하지 않으며(맨 끝에는 있을 수 있다) 분기 목적지나 분기 레이블도 없는(맨 앞에는 있을 수 있다) 시퀀스이다. 컴파일의 초기 단계 작업 중 하나는 프로그램을 기본 블록으로 나누는 것이다.

More Conditional Operations

slt $to, $s3, $s4 # $to =1 if $s3 < $s4

slti $to, $s2, 10 # $to =1 if $s2 < 10

Branch Instruction Design

blt, bqe를 구현하게 되면 클럭 속도가 느려지거나 이 명령 실행에 별도의 클럭 사이클이 더 필요하게 된다. 따라서 빠른 명령어 두개를 사용하는 것이 유리

Signed vs. Unsigned

signed comparsion(부호 있는 정수) : slt, slti

unsigned comparison : sltu, sltui

부호 없는 정수는 MSB가 1이면 그 누구보다 크고, 부호 있는 정수에는 MSB가 1인 수가 음수를 나타내며, MSB가 0인 어떤 양수보다 작다.

부호 없는 비교(sltu) x<y를 하면, x가 y보다 작은지뿐만 아니라 x가 음수인지도 검사가능

sltu $to, $s1, $t2 # $to=0 if $s1>=$t2 or $s1<0 , 0은 $zero 레지스터을 사용하면 된다.

beq $to, $zero, IndexOutOFBounds #if bad, goto Error

Procedure Calling

procedure은 이해하기 쉽고 재사용이 가능하도록 프로그램을 구조화하는 방법이다.

프로시저 단계

1. 프로시저가 접근할 수 있는 곳에 인수를 넣는다.

2. 프로시저로 제어를 넘긴다.

3. 프로시저가 필요로 하는 메모리 자원을 획득

4. 필요한 작업 수행

5. 호출한 프로그램이 접근할 수 있는 장소에 결과 값을 넣는다.

6. 프로시저는 프로그램 내의 여로 곳에서 호출될 수 있으므로 원래 위치로 제어를 돌려준다.

Register Usage

레지스터는 데이터를 저장하는 가장 빠른 장소이므로 가능한 한 많이 사용해야 함

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries -Can be overwritten by callee
- $s0 – $s7: saved -Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Procedure Call Instructions

procedure call : jump and link

jal procedureLabe1 지정된 주소로 점프하면서 동시에 다음 명령어의 주소를 $ra 레지스터에 저장

-> pc+4값을 $ra에다가 저장한다.

procedure return : jump register

jr $ra 레지스터 $ra에 저장되어 있는 주소로 점프(case/switch에서도 사용)

더많은 레지스터를 사용하려면 스택구조 사용

Non-Leaf Procedures

leaf 프로시저 -> 다른 프로시저를 호출하지 않는 프로시저

non-leaft 프로시저는 값이 보존되어야 할 모든 레지스터를 스택에 넣는 것이다.

(retrun address, Any arguments and temporaries needed after the call)

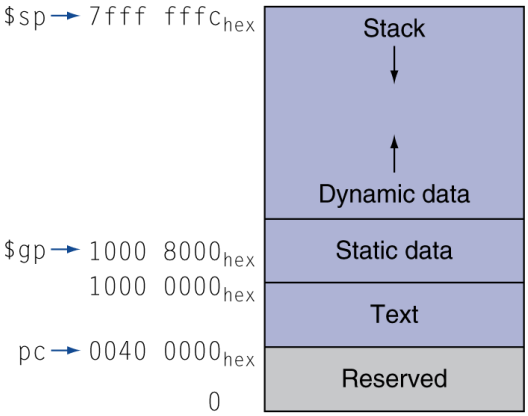
피피티 44쪽 중요! ->스택구조그려보기

Local Data on the Stack

procedure frame(activation record) -> 프로시저의 저장된 레지스터와 지역변수를 가지고 있는 스택 영역

MIPS에서는 fp(frame pointer)가 프로시저 프레임의 첫번째 워드를 가리킴. fp값으로 지역변수 참조가 간단, 또한 원상복구 시간절약

Memory Layout



C언어에서 mallco()는 힙에 공간을 할당한 후 이 공간을 가르키는 포인터를 결과 값으로 보내줌. free()는 포인터가 가르키는 힙공간 반납

$gp는 데이터에 쉽게 접글 할 수 있도록 주소가 초기화되어 있음.

Character Data

ASCII: 128 characters, Latin-1: 256 characters, Unicode: 32-bit character set

Byte/Halfword Operations -> ppt p48-50 다시보기

32-bit Constants

대부분 프로그램 상수는 16비트선에서 끝나는데 가끔 32비트까지 필요함.

MIPS는 레지스터의 상위16비트에 상수를 넣는 **lui** 명령어를 제공. 하위 16비트는 그 다음에 나오는 다른 명령으로 채움

lui rt, constant -> 밑에그림에서 lhi가 아니고 lui이다.

`lhi $s0, 61`	0000 0000 0111 1101 0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101 0000 1001 0000 0000

- 바이트(byte)를 로드하고 스토어하는 명령들: lb, lbu와 sb
- 하프워드(halfword)를 로드하고 스토어하는 명령들: lh, lhu, and sh ->16비트
- word을 로드하고 스토어하는 명령들 : lw, sw

Branch Addressing

PC-relative addressing ->Target address = PC + offset × 4 => 자주 생기는 일을 빠르게

PC 분기할 거리는 워드 단위로 한다. PC=레지스터+분기주소 -> 2^32까지 커지는 것 허용

OP	rs	rt	constant or address
6 bits	5bits	5bits	16bits

만일 프로그램에서 사용하는 모든주소가 이 16비트안에 들어간다면 프로그램은 2^16보다 커질 수 가 없다. 따라서 어떤 레지스터를 지정해서 그 값을 분기주소와 더하도록 한다.

Jump Addressing

OP	address
6 bits	26bits

명령은 항상 4의 배수의 주소에서 시작한다.(바꿔말해 워드정렬되어있다) 그러므로 32비트 명령 주소의 하위 2비트는 항상 "00"이다. 26비트의 target을 왼쪽으로 2자리 이동시키명 28비트 크기의 워드 정렬된 주소가된다.(다시말해 하위 2비트가 "00"이된다) 이제 주소의 상위 4비트를 채우는 일만 남았다. 이 4비트는 PC의 상위 4비트로부터 가져온다. PC 상위 4비트를 28비트에 붙여서 32비트 크기의 주소를 결과적으로 만들어 낸다.

80000	0	0	19	9	4	0	Loop: sll $t1, $s3, 2
80004	0	9	22	9	0	32	add $t1, $t1, $s6
80008	35	9	8	0			lw $t0, 0($t1)
80012	5	8	21	2			bne $t0, $s5, Exit
80016	8	19	19	1			addi $s3, $s3, 1
80020	2	20000					j Loop
80024							Exit: …

-> bne을 살펴보면 Exit의 주소를 2로표시했는데 이거는 다음명령어의 주소 80016+2*4한 값으로 간다는 뜻이다. **현재 명령어 기준이 아니고 다음 명령어를 기준으로 한 상대적 위치가 명령어에 표시된다.**

Branching Far Away

*beq $s0,$s1, L1*은 L1의 크기가 i-format이라서 16비트까지 밖에 표현이 안된다. 그래서 더 크게 표현할려면 jump명령어의 포맷으로 하면 26비트까지 표현가능하다.

```
=>      bne $s0,$s1, L2
        j L1
L2:     ...
```

MIPS 주소지정 방식(Addressing Mode Summary)

1. 수치 주소지정 : 피연산자는 명령어 내에 있는 상수이다.

2. 레지스터 주소지정 : 피연산자는 레지스터이다.

3. 베이스 또는 변위 주소지정 : 메모리 내용이 피연산자이다.

메모리 주소는 레지스터와 명령어 내의 상수를 더해서 구한다.

4. PC 상대 주소지정 : PC값과 명령어 내 상수의 합을 더해서 주소를 구한다.

5. 의사직접(pseudodirect) 주소지정 : 명령어 내의 26비트를 PC의 상위 비트들과 연접하여 점프 주소를 구한다.

MIPS에서 조건문 분기 주소범위(k=1024) => 분기문 전 128k~ 후 128K까지

MIPS에서 j와 jal 명령을 수행할 수 있는 주소의 범위 => 주소의 상위 4비트와 PC와 같은 256M 블록 내 임의 장소

2.10 병렬성과 명령어: 동기화

-> 기본적 개념은 OS에서 배움

동기화를 위해 한쌍의 명령어를 갖느데 두번째 명령어는 한쌍의 명령어가 마치 원자적인 것처럼 실행되었는지를 나타내는 값을 반환해야함.

MIPS에서는 이러한 명령어 쌍으로 load linked라 불리는 특수 적재 명령어와 store conditional이라 불리는 특수 저장 명령어가 있다.

```
try:  add $t0,$zero,$s4 ;copy exchange value
      ll  $t1,0($s1)    ;load linked
      sc  $t0,0($s1)    ;store conditional
      beq $t0,$zero,try ;branch store fails
      add $s4,$zero,$t1 ;put load value in $s4
```

2.11 프로그램 번역과 실행

컴파일러 : C 프로그램을 어셈블리 언어로 바꾼다.

어셈블리 : 원래는 없는 명령어를 어셈블러가 독자적으로 제공가능

의사 명령어 (하드웨어가 지원하지 않는 어셈블리 언어 명령어를 마치 실제 있는 것처럼 어셈블러가 처리하는 명령어)

어셈블리 언어 프로그램을 구성하는 각 명령어를 이진수로 바꾸기 위해서는 레이블에 해당하는 주소를 모두 알아야함 -> 이러한 레이블을 심볼 레이블에 저장함.

심볼 레이블 : 레이블 이름을 명령어가 기억된 메모리 워드의 주소와 짝지어 주는 테이블

링커 : 따로 따로 어셈블된 기계어 프로그램을 하나로 연결해주는 일을 한다.

1. 코드와 데이터 모듈을 메모리에 심벌 형태로 올려 놓는다.

2. 데이터와 명령어 레이블의 주소를 결정한다

3. 외부 및 내부 참조를 해결한다.

링커는 실행파일(executable file)을 생성

로더 : 목적 프로그램을 메인 메모리에 적재해서 실행 할 수 있게 하는 시스템 프로그램

동적 링크 라이브러리 : 실행시에 프로그램과 링크되는 라이브러리 루틴

예제 및 스스로 점검하기

p69-70, p73-75, p77, p81-83, **p86**, p89-91, p95, **p96**-101, p104, **p107,** p112-113, p116-121 ,p124 , p128

책 2.19.(1-3), 2.39, 2.31