

제 8강. 1. MIPS Addressing for 32-Bit Immediates and Addresses 2. Parallelism and Instructions: Synchronization 3. Translating and Starting a Program

2.10 ARM의 32비트 수치를 위한 주소지정 및 복잡한 주소지정방식

MIPS - 16bit immediate is sufficient

32bit constant를 위해선

lhi rt,

- 왼쪽 rt의 16비트에 복사됨

ori

- 오른쪽 16bit

32비트 수치 피연산자

DP형식의 12비트 operand2 필드는 오른쪽의 상수 필드 8비트와 오른쪽 회전 필드 4비트로 나눈어진다.

→ 이 방법을 이용하면 다음 값을 갖는 부호 없는 정수는 어느 것이라도 표현할 수 있다.



$$X * 2^{2i}$$

Branch Addressing

- Branch 연산은
 - Opcode, 2 registers, target address

- 대부분의 브랜치 타겟은 브랜치 가까이 있다
 - 앞, 뒤
 -

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

- PC(Program Counter)-relative addressing
 - Target address = PC + offset*4
 - PC는 이미 4만큼 증가돼 있다.

Jump Addressing

op	constant or address
6bits	26bits

- (Pseudo) Direct jump addressing
 - Target Address = PC.... + (address * 4)

현재 PC의 31번비트~28번비트까지

Target Addressing Example

Loop code from earlier example

location 80000

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

$80016 + 2 * 4 = 80024$

$0000 * 20000 * 4 = 80000$

Branching Far away

브랜치 타겟이 너무 멀어서 16bit offset을 넘으면

Example

```
beq $s0,$s1, L1
      ↓
bne $s0,$s1, L2
j L1
L2: ...
```

branch 명령어 대신 jump 명령어를 씀

jump는 26bit를 씀

데이터 전송 명령어의 복잡한 주소 지정

Addressing Mode Summary ★ 암기!

- 여러 개의 주소 지정 형태 : 주소지정방식 (addressing mode)

수치 변위

- 레지스터 변위 (Register Offset) : 베이스 레지스터에 상수가 아니라 다른 레지스터를 더함. 이 방식은 한 레지스터는 인덱스를 다른 레지스터는 배열의 시작주소를 갖고 있어 인덱스로 배열을 찾아갈 때 유용
- 스케일된 레지스터 변위 (Scaled Register Offset) : 레지스터를 먼저 자리이동한 후, 베이스 레지스터에 더함. 배열 인덱스를 2비트 왼쪽 자리로 이동해 바이트 주소로 변환하는 데 유용
→ 2.14절
- 수치 변위 인덱스
- 수치 변위 포스트 인덱스
- 레지스터 변위 프리인덱스
- 스케일 레지스터 변위 프리인덱스

7. 레지스터 변위 포스트인덱스
8. Immediate Addressing (수치값 주소지정방식) 피연산자가 명령어 내의 상수
ex. ADD r2, r0, #5
9. Register Addressing (레지스터 주소지정방식) : 피연산자는 레지스터
ex. ADD r2, r0, r1
10. Scaled Register Addressing (스케일된 레지스터 주소지정방식) : 레지스터 피연산자가 먼저 자리이동한다.
ADD r2, r0, r1, LSL #2
11. PC-Relative Addressing (PC 상대 주소지정방식) : PC값과 명령어의 상수값을 더한 값이 분기 주소가 됨. ex. BEQ 1000
12. Base Addressing

2.11 병렬성과 명령어:동기화

2 processors가 하나의 메모리를 공유

- P1 writes, P2 reads
- Data race 위험. P1 and P2가 동기화되지 않았을 때
 - order of accesses에 영향 받은 결과

하드웨어 서포트는 요구한다.

- Atomic read/write memory operation ⇒ **spin lock**
- read and write 사이에 다른 location의 접근을 허용하지 않는다.

single 연산이 될 수 있다.

- atomic swap of register ↔ memory
- Or an atomic pair of 연산

↳ 2개 연산이 하나로 되게끔?

skip!

MIPS에서의 동기화

- Load linked : rt, offset(rs)

2.12 프로그램 번역과 실행

C프로그램 → (컴파일러) → 어셈블리 언어 프로그램 → (어셈블러) → 목적 코드 : 기계어 모듈 , 목적 코드 : 라이브러리 루틴(기계어) → (링커) → 실행 코드 : 기계어 프로그램 → (로더) → 메모리

└ Static Linking ┐

의사 명령어 : 어셈블러 Pseudo명령어

대부분의 어셈블러 명령어는 기계 명령어를 1대1로 대표한다.

move \$t0, \$t1 → add \$t0, \$zero, \$t1

blt \$t0, \$t1, L → slt \$at, \$t0, \$t1

bne \$at, \$zero, L

\$at (register1) : assembler temporary

어셈블러

- 어셈블러의 주된 임무 : 어셈블리 프로그램을 기계어로 번역하는 일
 - 어셈블리 프로그램 → 목적 파일(object file)로 바꿈
 - 목적 파일 ⊃ 기계어 명령어, 데이터, 명령어를 메모리에 적절히 배치하기 위한 각종 정보들 혼합
 - 심볼 테이블 에 분기나 데이터 전송 명령에서 사용된 모든 레이블을 저장
- UNIX 시스템의 목적 파일 구분
 - 목적 파일 헤더 : 목적 파일을 구성하는 각 부분의 크기와 위치 서술
 - 텍스트 세그먼트 : 기계어 코드가 들어 있음
 - 정적 데이터 세그먼트 : 프로그램 수명 동안 할당되는 데이터

- UNIX 프로그램 실행이 끝날 때까지 계속 할당되는 **정적 데이터**와 프로그램의 요구에 따라 커졌다 작아졌다 하는 **동적 데이터**
 - 재배치 정보 : 프로그램이 메모리에 적재될 때 절대주소에 의존하는 명령어와 데이터 워드 표시
 - 심볼 테이블 : 외부 참조와 같이 아직 정의되지 않고 남아 있는 레이블들을 저장
 - 디버깅 정보 : 각 모듈이 어떻게 번역되었는지에 대한 간단한 설명, 디버거는 이 정보를 이용해서 기계어와 C 소스 파일을 연관짓고 자료구조를 판독한다.

링커

바뀌지 않는 루틴들을 계속 수정할 필요 없게 각 프로시저를 따로따로 컴파일, 어셈블함

링커의 동작

1. 코드와 데이터 모듈을 메모리에 심볼 형태로 올려 놓는다
2. 데이터와 명령어 레이블의 주소를 결정
3. 외부 및 내부 참조를 해결

링커의 역할

- 각 목적 모듈의 재배치 정보와 심볼 테이블을 이용해 미정의 **레이블의 주소를 결정**
- 분기 명령어, 점프 명령어, 데이터 주소 등에 나타나는 구주소를 신주소로 바꾸는 일을 하므로 에디터와 유사함
- 프로그램 전체를 다시 컴파일하고 어셈블하는 대신 링커를 써서 번역된 모듈을 연결하면 시간이 절약
- 외부 참조를 모두 해결하고 나면 **각 모듈의 메모리 주소를 결정**
- 각 파일을 독립적으로 어셈블하기 때문에, 어셈블러는 어떤 모듈의 명령어와 데이터가 다른 모듈과 비교해서 어떤 위치에 있게 될는지 알 수 없다.
- 링커가 모듈을 메모리에 적재할 때 절대참조는 모두 실제 위치에 해당하는 값으로 재설정돼야 함. (location dependency 정보는 필요 없음)
- **실행 파일을 생성**
 - 미해결된 참조는 없고, 목적 파일과 같은 형식을 갖는다.

로더

디스크에 있는 실행 파일을 메모리에 넣고 이를 시작시킴

진행 순서

1. 실행 파일 헤더를 읽어서 텍스트와 데이터 세그먼트의 크기를 알아냄
2. 텍스트와 데이터가 들어갈 만한 주소공간을 확보
3. 실행 파일의 명령어와 데이터를 메모리에 복사
4. 주 프로그램에 전달해야 할 인수가 있으면 이를 스택에 복사
5. 레지스터를 초기화하고 스택 포인터는 사용 가능한 첫 주소를 가리키게 함
6. 기동 루틴(start-up routine)으로 점프. 이 기동 루틴에서는 인수를 인수 레지스터에 넣고 프로그램 주 루틴을 호출. 주 프로그램에서 기동 루틴으로 복귀하면 exit 시스템 호출을 사용하여 프로그램 종료.

동적 링크 라이브러리 DLL

오직 사용될 때만 link/load 가 링킹되게 하는 것

프로그램 실행 전에는 라이브러리가 링크되지도 적재되지도 않음

대신, 프로그램과 라이브러리 루틴은 전역적 프로시저의 위치와 이름에 대한 정보를 추가로 가지고 있음.

- 프로시저 코드가 재배치될 수 있도록 작성해야 함
- 실행 파일의 크기가 커지는 것 방지
- 자동으로 new version을 픽함
- 처음에는 아주 복잡한 과정을 거침

Lazy Linkage [그림 2.22]

stub : link와 로드에 대한 위치 정보

linker/loader code에 실제 위치 정보 있음

크기가 커지지 않고, 새로운 라이브러리를 만들어도 다시 컴파일할 필요 없음

→ Java 나중에 배움