2장

지침: 컴퓨터 언어

명령어 세트

- 컴퓨터의 명령어 레퍼토리
- 컴퓨터마다 명령어 세트가 다릅니다.
  - 하지만 많은 측면에서 공통점이 있습니다.
- 초기 컴퓨터는 명령어 세트가 매우 단순했습니다.
  - 간소화된 구현

- 많은 최신 컴퓨터에는 간단한 명령어 세트도 있습니다.

RISC-V 명령어 세트

- 책 전체에서 예시로 사용
- UC 버클리에서 개방형 ISA로 개발되었습니다.
- 이제 RISC-V 재단(riscv.org)에서 관리합니다.
- 많은 최신 ISA의 특징
- 유사 ISA는 임베디드 코어 시장에서 큰 점유율을

차지하고 있습니다.

- 소비자 가전, 네트워크/스토리지 분야 애플리케이션 장비, 카메라, 프린터, ...

산술 연산

- 덧셈과 뺄셈, 세 개의 피연산자
  - 두 개의 출처와 하나의 목적지

추가 `A, B, C // A`가 `B + C`를 가져옵니다.

- 모든 산술 연산은 다음과 같은 형식을 갖습니다.
- *디자인 원칙 1:* 단순함이 규칙성을 선

호합니다.

- 규칙적으로 구현하면 구현이 더 간단해집니다.
- 단순성으로 더 낮은 비용으로 더 높은 성능 제공

산술 예제

- C 코드:

```
f = (g + h) - (i + j);
```

- 컴파일된 RISC-V 코드:

```
추가 T0, G, H // 임시 T0 = G + H 추
가 T1, I, J // 임시 T1 = I + J 서브
F, T0, T1 // F = T0 - T1
```

피연산자 등록

- 산술 명령어는 레지스터 피연산자를 사용합니다.

- RISC-V에는 32 × 64비트 레지스터 파일이 있습니다
  - 자주 액세스하는 데이터에 사용
  - 64비트 데이터를 "더블워드"라고 합니다.
    - 32 x 64비트 범용 레지스터 x0~x31

- 32비트 데이터를 "워드"라고 합니다.
  - *디자인 원칙 2:* 작을수록 빠름
  - 참조: 메인 메모리: 수백만 개의 위치

RISC-V 레지스터

- x0: 상수 값 0
- x1: 반환 주소
- x2: 스택 포인터
- x3: 글로벌 포인터
- x4: 스레드 포인터
- X5 - X7, X28 - X31: 임시직
- x8: 프레임 포인터

- X9, X18 - X27: 저장된 레지스터
- X10 - X11: 함수 인수/결과
- x12 - x17: 함수 인자

레지스터 피연산자 예제

- C 코드:

```
f = (g + h) - (i + j);
```

  - F, ..., J in X19, X20, ..., X23

- 컴파일된 RISC-V 코드:

```
추가 x5, x20, x21
추가 x6, x22, x23
sub x19, x5, x6
```

메모리 피연산자

- 복합 데이터에 사용되는 메인 메모리
  - 배열, 구조, 동적 데이터
  - 산술 연산을 적용하려면
    - 메모리에서 레지스터로 값 로드
    - 레지스터에서 메모리로 결과 저장
- 메모리는 바이트 주소가 지정됩니다.
  - 각 주소는 8비트 바이트를 식별합니다.

- RISC-V는 리틀 엔디안
  - 단어의 최하위 바이트 이상 주소
  - *참조*: 빅 엔디안: 가장 중요한 바이트 이상의 주소
- RISC-V는 메모리에서 단어를 정렬할 필요가 없습니다.
  - 다른 ISA와 달리

메모리 피연산자 예제

- C 코드:

```
A[12] = h + A[8];
```

  - x21의 h, x22의 A 기본 주소

- 컴파일된 RISC-V 코드:

  - 인덱스 8은 64의 오프셋이 필요합니다.

    - 이중 단어당 8바이트

```
ld      x9, 64(x22)
추가    x9, x21, x9
sd      x9, 96(x22)
```

레지스터 대 메모리

- 레지스터는 메모리보다 액세스 속도가 빠릅니다.
- 메모리 데이터에 대한 작업에는 로드 및 저장
  - 실행할 추가 지침
- 컴파일러는 가능한 한 변수에 레지스터를 사용해야 합니다.

- 자주 사용하지 않는 변수에 대해서만 메모리로 유출합니다.
- 등록 최적화가 중요합니다!

즉시 피연산자

- 명령어에 지정된 상수 데이터

```
addi x22, x22, 4
```

- 일반 케이스를 빠르게 만들기
  - 작은 상수는 일반적입니다.
  - 즉시 피연산자가 로드 명령을 피합니다.

부호 없는 이진 정수

- n비트 숫자가 주어지면

$$x = x_{n-1} 2^{n-1} + x_{n-2} 2^{n-2} + \boxtimes + x 2^{1} + x 2^{0}$$

- 범위: 0 ~ $^{+2n}$ - 1
- 예
  - $0000\ 0000 \ldots 0000\ 1011_2$
    $= 0 + \ldots + {}^{1\times23} + {}^{0\times22} {}^{+1\times21} {}^{+1\times20}$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- 64비트 사용: 0 ~ +18,446,774,073,709,551,615

2-보완 부호 있는 정수

- n비트 숫자가 주어지면

$$x = -x_{n-1} 2^{n-1} + x_{n-2} 2^{n-2} + \boxtimes + x\,2^{1} + x\,2^{0}$$

- 범위: $-2n - 1 \sim +2n - 1 - 1$
- 예
  - $1111\ 1111 \ldots 1111\ 1100_2$
    $= -1\times231 + 1\times230 + \ldots + 1\times22 + 0\times21 + 0\times20$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

- 64비트 사용: −9,223,372,036,854,775,808

  9,223,372,036,854,775,807원

2-보완 부호 있는 정수

- 비트 63은 부호 비트
  - 음수인 경우 1
  - 음수가 아닌 숫자의 경우 0
- $-(^{-2n - 1})$은 표현할 수 없습니다.
- 음수가 아닌 숫자는 부호가 없는 2소수 표현이 동일합니다.
- 몇 가지 특정 숫자

- 0: 0000 0000 ... 0000
- -1: 1111 1111 ... 1111
- 가장 부정적: 1000 0000 ... 0000
- 가장 긍정적: 0111 1111 ... 1111

서명된 부정

- 보완 및 추가 1
  - 보수는 1 → 0, 0 → 1을 의미합니다.

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- 예: +2 부정

- +2 = 0000 0000 ... 0010_{two}
- -2 = 1111 1111 ... 1101_{two} + 1
  = 1111 1111 ... 1110_{two}

서명 확장

- 더 많은 비트를 사용하여 숫자 표현하기
  - 숫자 값 보존
- 왼쪽에 부호 비트를 복제합니다.
  - 참조: 부호 없는 값: 0으로 확장하기
- 예시: 예: 8비트~16비트
  - +2: 0000 0010 => 0000 0000 0000 0010
  - -2: 1111 1110 => 1111 1111 1111 1110

- RISC-V 명령어 집합에서
  - `lb`: 서명-로드된 바이트 확장
  - `lbu`: 제로 확장 로드된 바이트

지시 사항 표현

- 인스트럭션은 바이너리로 인코딩됩니다.
  - 머신 코드라고 합니다.
- RISC-V 지침
  - 32비트 명령어로 인코딩
  - 연산 코드(옵코드), 레지스터 번호 등을 인코딩하는 소수의 형식, ...

- 규칙성!

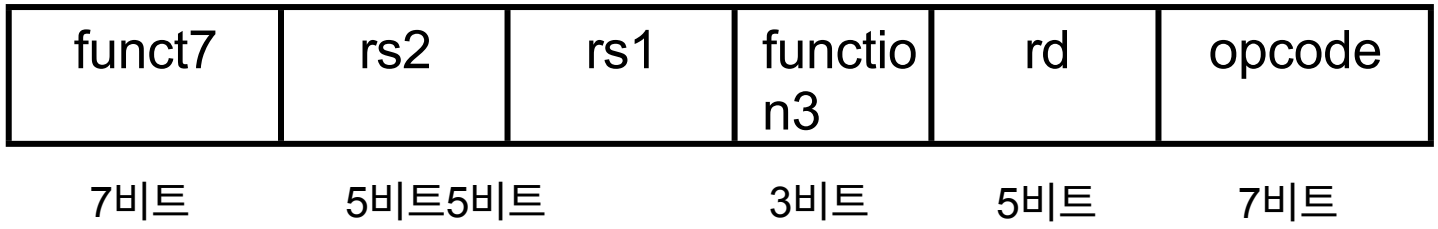
16진수

- 기본 16
  - 비트 문자열의 간결한 표현
  - 16진수당 4비트

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- 예: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-포맷 지침



- 지침 필드
  - OP코드: 작업 코드
  - rd: 목적지 등록 번호
  - function3: 3비트 함수 코드(추가 옵코드)

- RS1: 첫 번째 소스 레지스터 번호
- RS2: 두 번째 소스 레지스터 번호
- function7: 7비트 기능 코드(추가 연산 코드)

R 형식 예제

funct7	rs2	rs1	function3	rd	opcode
7비트	5비트	5비트	3비트	5비트	7비트

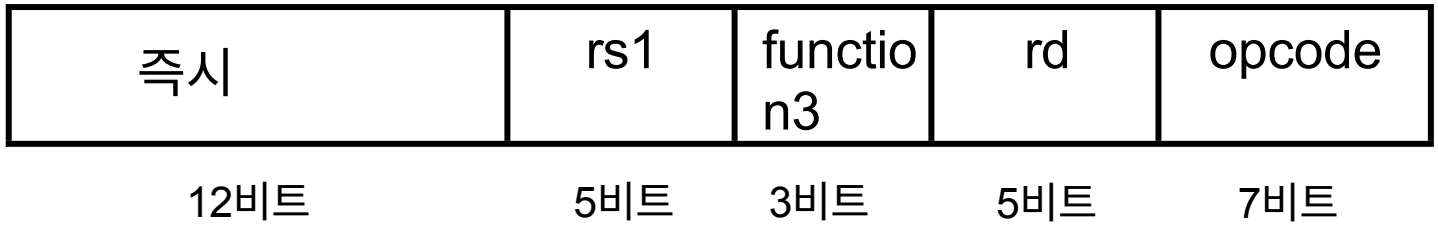
추가 `X9,X20,X21`

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

0000 0001 0101 1010 0000 0100 1011 0011 = $=_{two}$

$015A04B3_{16}$

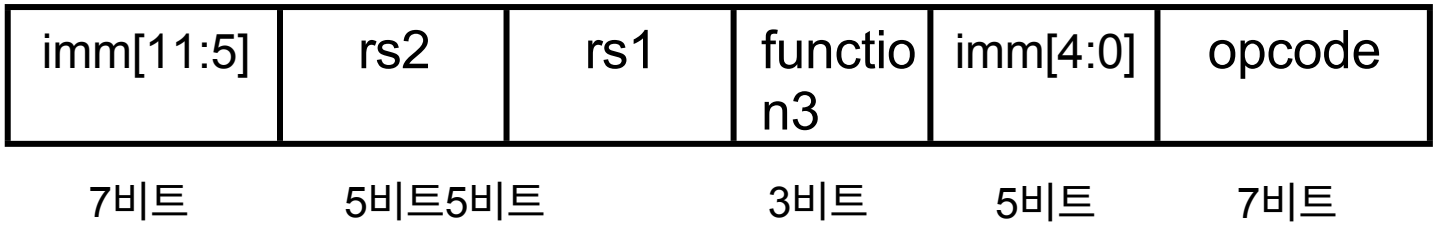
RISC-V I-포맷 지침



- 즉각적인 산술 및 로드 지침
  - RS1: 소스 또는 기본 주소 레지스터 번호
  - 즉시: 상수 피연산자 또는 기본 주소에 추가되는 오프셋
    - 2초 보완, 부호 확장
- *디자인 원칙 3:* 좋은 디자인은 좋은 것을 요구합니다. 타협

- 형식이 다르면 디코딩이 복잡해지지만 32비트 명령어를 균일하게 허용합니다.
- 형식을 최대한 유사하게 유지하세요.

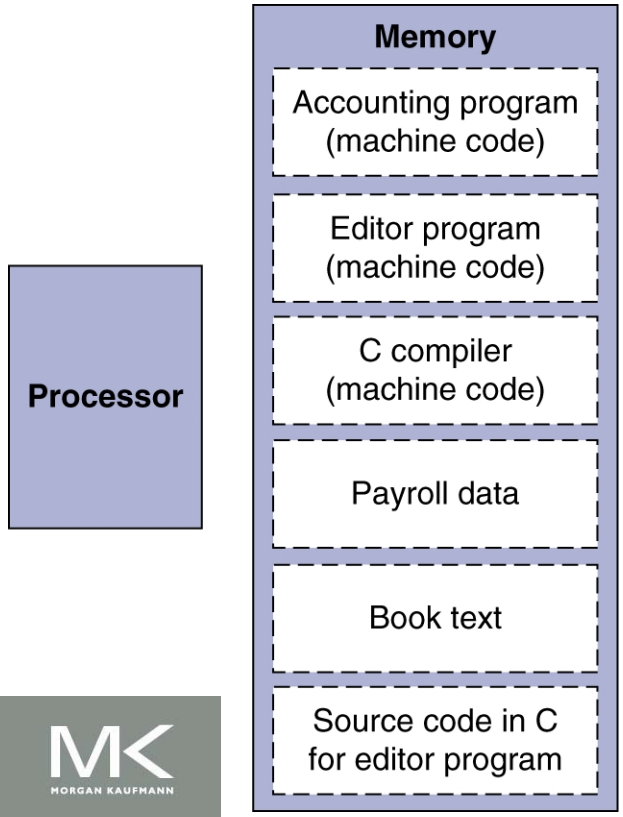
RISC-V S-포맷 지침



- 스토어 지침에 대한 다양한 즉각적인 형식
  - RS1: 기본 주소 등록 번호
  - RS2: 소스 피연산자 레지스터 번호
  - 즉시: 기본 주소에 오프셋 추가
    - RS1 및 RS2 필드가 항상 같은 위치에 있도록 분할

저장된 프로그램 컴퓨터

큰 그림



- 데이터와 마찬가지로 바이너리로 표시되는 명령어
- 메모리에 저장된 지침 및 데이터
- 프로그램에서 작동할 수 있는 프로그램
  - 예: 컴파일러, 링커, ...

- 바이너리 호환성을 통해 컴파일된 프로그램이 다른 컴퓨터에서 작동할 수 있습니다.
  - 표준화된 ISA

논리 연산

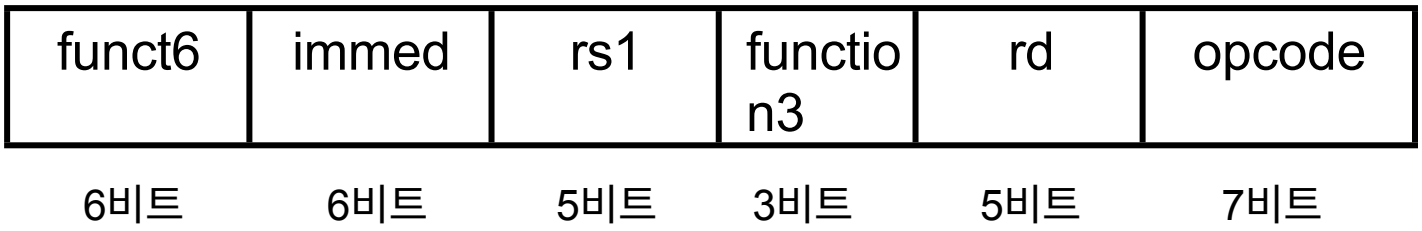
- 비트 단위 조작 지침

운영	C	Java	RISC-V
왼쪽으로 이동	<<	<<	`slli`
오른쪽으로 이동	>>	>>>	`srli`
비트 단위 AND	&	&	그리고
비트 단위 또는	|	|	또는, `ori`
비트별 XOR	^	^	`xor, xori`

비트 단위로 NOT	~	~	

- NOT RD, RS : XORI RD, RS, -1에 의해 구현됨
- 단어에서 비트 그룹을 추출하고 삽입하는 데 유용합니다.

교대 근무



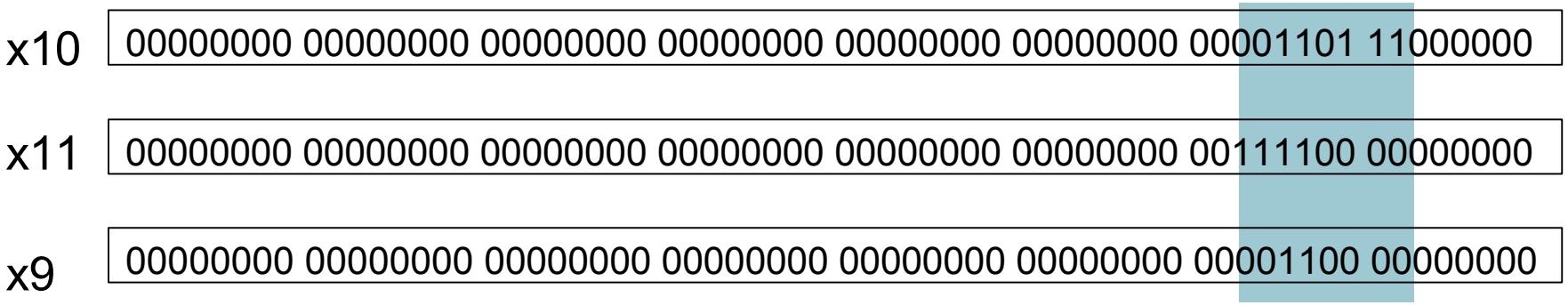
- 즉시: 이동해야 할 포지션 수
- 논리적으로 왼쪽으로 이동
  - 왼쪽으로 이동하여 0비트로 채우기
  - SLLI에 *I* 비트 곱하기 *2I*

- 오른쪽 논리 전환
  - 오른쪽으로 이동하여 0비트로 채우기
  - *SRLI* 바이트를 2^{I}로 나눈 값(부호 없는 경우에만 해당)

AND 연산

- 단어의 비트를 마스킹하는 데 유용합니다.
  - 일부 비트를 선택하고 다른 비트를 0으로 지웁니다.

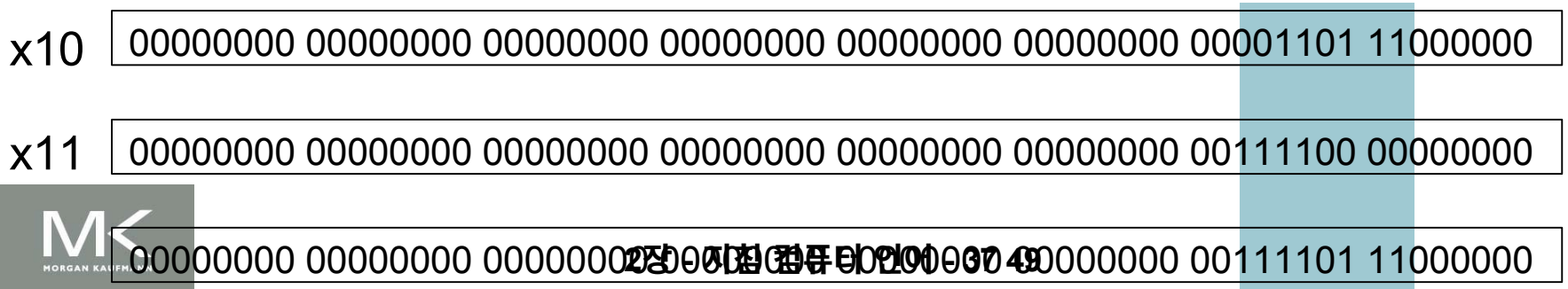
및 `x9,x10,x11`



수술실 운영

- 단어에 비트를 포함할 때 유용합니다.
  - 일부 비트를 1로 설정하고 다른 비트는 변경하지 않습니다.

또는 `x9,x10,x11`

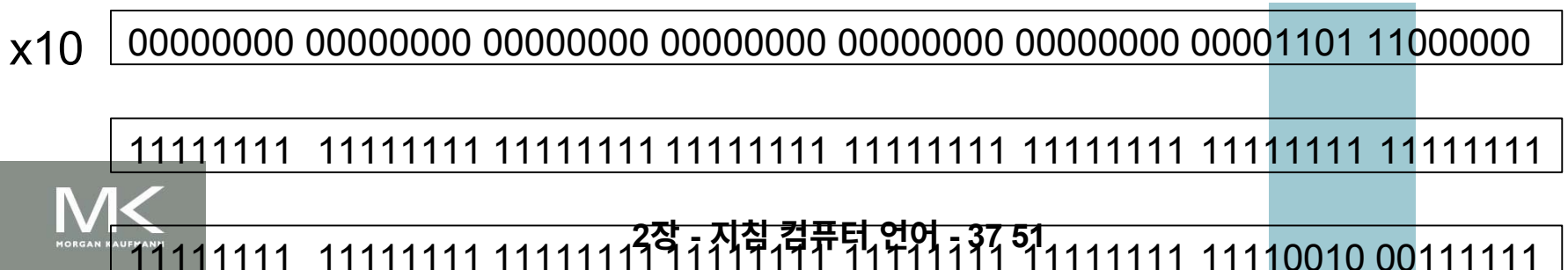


x9

XOR 연산

- XOR은 비트가 같으면 0을, 다르면 1을 생성합니다.
  - x 또는 111...111 : NOT과 동일합니다.

```
xor x9,x10,x12 // 작동하지 않음
```



x12

x9

조건부 연산

- 조건이 참이면 레이블이 지정된 명령어로 분기합니다.
  - 그렇지 않으면 순차적으로 계속 진행합니다.
- `beq RS1, RS2, L1`
  - IF (RS1 == RS2) L1 레이블이 지정된 명령어로 분기합니다.

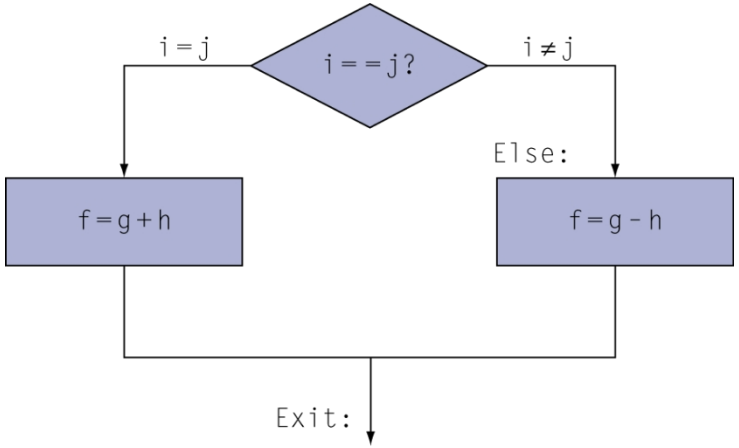
- `BNE RS1, RS2, L1`
  - IF (RS1 != RS2) L1 레이블이 지정된 명령어로 분기합니다.

If 문 컴파일하기

- C 코드:

```
if (i==j) f = g+h;
else f = g-h;
```

  - F, G, ... IN X19, X20, ...

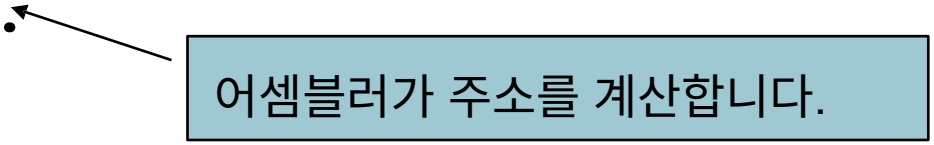


- 컴파일된 RISC-V 코드:

```
bne x22, x23, 기타

x19, x20, x21 추가
```

```
        beq x0,x0,Exit // 무조건 Else:
sub x19, x20, x21
종료: ...
```



루프 문 컴파일하기

- C 코드:

```
while (save[i] == k) i += 1;
```

  - X22의 I, X24의 K, X25의 저장 주소

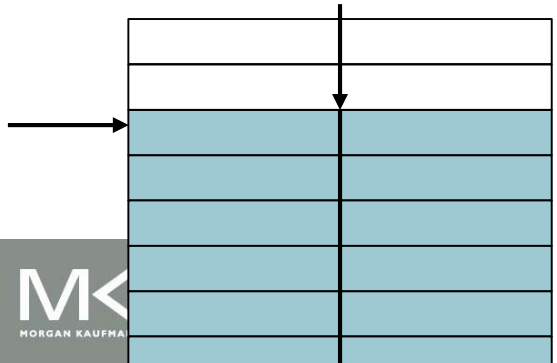
- 컴파일된 RISC-V 코드:

```
Loop: slli x10, x22, 3
      추가  x10, x10, x25
      ld    x9, 0(x10)
      bne   x9, x24, 출구
```

```
        addi x22, x22, 1
        beq  x0, x0, 루프
종료합  ...
니다:
```

기본 블록

- 기본 블록은 다음과 같은 명령어 시퀀스입니다.
  - 임베디드 브랜치 없음(끝부분 제외)
  - 분기 대상 없음(처음 제외)



- 컴파일러는 최적화를 위한

기본 블록을 식별합니다.

- 고급 프로세서는 기본 블록의 실행을 가속화할 수 있습니다.

더 많은 조건부 연산

- `BLT RS1, RS2, L1`
  - IF (RS1 < RS2) L1 레이블이 지정된 명령어로 분기합니다.
- `BGE RS1, RS2, L1`
  - IF (RS1 >= RS2) L1 레이블이 붙은 명령어로 분기합니다.

- 예
  - IF (A > B) A += 1;
  - A는 X22, B는 X23

```
bge, x22, 종료                    //branch if b >= a
addi x22, x22, 1
```

종료합니다:

서명 대 미서명

- 서명 비교: BLT, BGE
- 부호 없는 비교: bltu, bgeu
- 예
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 //` 서명

- -1 < +1

- `x22 > x23 //` 서명되지 않음
  - +4,294,967,295 > +1

프로시저 호출

- 필요한 단계

1. 레지스터 x10~x17에 파라미터를 배치합니다.
2. 절차로 제어권 이전
3. 절차를 위한 스토리지 확보
4. 절차의 작업 수행
5. 발신자 등록에 결과 배치

6. 통화 장소로 돌아가기(x1의 주소)

절차 호출 지침

- 프로시저 호출: 점프 및 링크

  `jal x1,` 프로시저레이블

  - x1에 입력한 다음 명령어의 주소
  - 대상 주소로 이동

- 프로시저 반환: 점프 및 링크 등록

  `JALR X0, 0(X1)`

- jal과 비슷하지만 0 + 주소로 점프합니다.
- x0을 RD로 사용(x0은 변경 불가, 리턴 주소 없음)
- 계산된 점프에도 사용 가능
  - 예: 케이스/스위치 문의 경우

리프 프로시저 예시

- C 코드:

```
long long int leaf_example (
    롱 롱 인트 G, 롱 롱 인트 H,
    롱 롱 인트 I, 롱 롱 인트 J) { 롱 롱 인트 F;
  f = (g + h) - (i + j);
  f를 반환합니다;
}
```

  - 인자 g, ..., j in x10, ..., x13 (함수 인자 reg)

- f in x20(저장된 레그)
- 임시 X5, X6
- 스택에 x5, x6, x20을 저장해야 함(일반적으로 스택에 저장되지 않는 임시 레그)

리프 프로시저 예시

- RISC-V 코드:

```
leaf_example:
   addi  sp,sp,-24
   sd    X5,16(SP)
   sd    x6,8(SP)
   sd    x20,0(SP)
   추가  x5,x10,x11
   추가  x6,x12,x13
   sub   x20,x5,x6
   addi  x10,x20,0
   ld    x20,0(SP)
```

스택에 x5, x6, x20 저장

x5 = g + h

x6 = i + j

f = x5 - x6

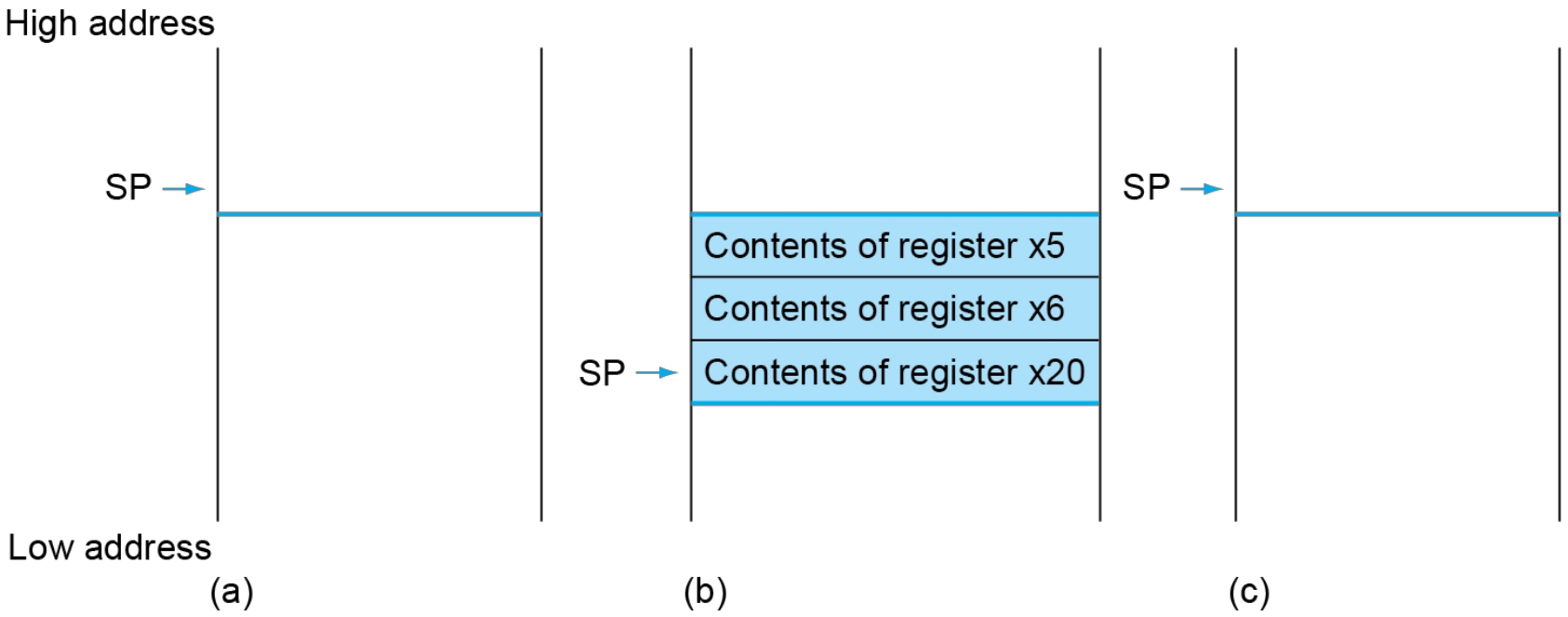
복사 F를 반환 레지스터 X10으로

스택에서 x5, x6, x20 복원

```
ld   x6,8(SP)
ld   X5,16(SP)
addi sp,sp,24
jalr x0,0(x1)
```

발신자에게 반환, x1: 반환 주소

스택의 로컬 데이터


High address
SP
Low address
(a)
Contents of register x5
Contents of register x6
SP
Contents of register x20
(b)
SP
(c)

사용 등록

- X5 - X7, X28 - X31: 임시 레지스터
  - 발신자가 보존하지 않음
- X8 - X9, X18 - X27: 저장된 레지스터
  - 사용되면 호출자가 저장하고 복원합니다.

비리프 절차

- 다른 프로시저를 호출하는 프로시저
- 중첩 호출의 경우 호출자는 스택에 저장해야 합니다:
  - 반환 주소
  - 통화 후 필요한 인수 및 임시 항목

- 통화 후 스택에서 복원

비리프 프로시저 예시

- C 코드:

```
긴 긴 int                    사실 (긴 long int n)
{
   if (n < 1) 1을 반환합니다
                ;
   그렇지 않으면  n * fact(n - 1);
   반환
}
```

- 인자 n(x10)
- 결과 x10

비리프 프로시저 예시

- RISC-V 코드:

사실:

```
    ADDI SP,SP,-16
    sd   x1,8(sp)
    sd   x10,0(sp)
    ADDI X5,X10,-1
    BGE  x5,x0,L1
    ADDI x10,x0,1
    addi sp,sp,16
    jalr x0,0(x1)
L1: addi x10,x10,-1
```

스택에 반송 주소와 n 저장

x1: 반환 주소

x10: 인자 n

x5 = n - 1

n >= 1이면 L1로 이동합니다.

그렇지 않으면 반환 값을 1로 설정합니다.

팝 스택, 값을 복원하는 데 신경 쓰지 않아도 됩니다.

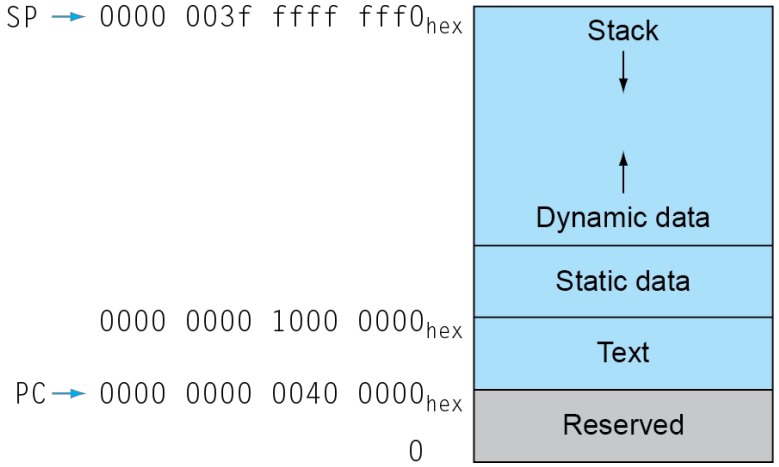
반환

n = n - 1

`jal`	`x1,사실`	호출 사실(n-1)
`addi`	`x6,x10,0`	fact(n - 1)의 결과를 x6으로 이동합니다.
`ld`	`x10,0(SP)`	발신자 번호 복원
`ld`	`x1,8(SP)`	발신자의 반송 주소 복원
`addi`	`sp,sp,16`	팝 스택
`mul`	`x10,x10,x6`	반환 n * fact(n-1)
`jalr`	`x0,0(x1)`	반환

메모리 레이아웃

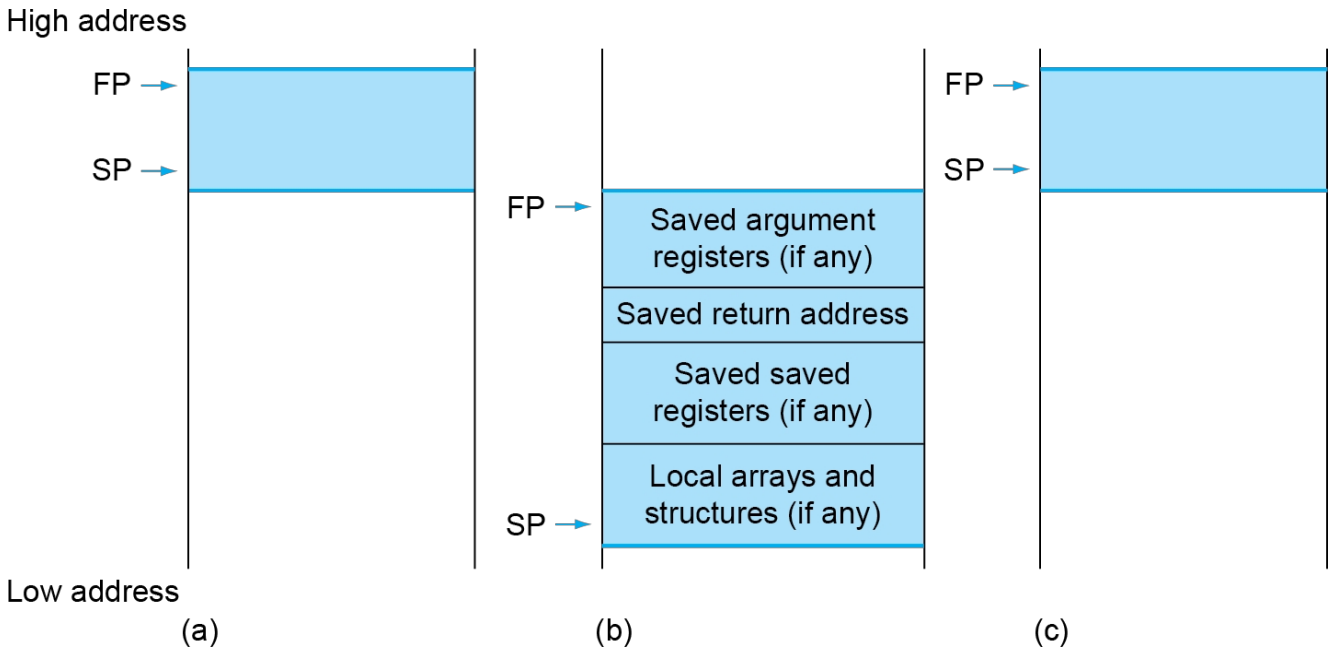
- 텍스트: 프로그램 코드
- 정적 데이터: 전역 변수
  - 예: C의 정적 변수, 상수 배열 및 문자열
  - 이 세그먼트에 ± 오프셋을 허용하는 주소로 초기화된



x3(전역 포인터)

- 동적 데이터: 힙
  - 예: C의 malloc, Java의 새로운 기능
- 스택: 자동 저장

스택의 로컬 데이터



- 호출자가 할당하는 로컬 데이터
  - 예: C 자동 변수

- 절차 프레임(활성화 기록)
  - 일부 컴파일러에서 스택 저장소 관리에 사용

문자 데이터

- 바이트 인코딩 문자 집합
  - ASCII: 128자
    - 그래픽 95, 제어 33
  - 라틴어-1: 256자
    - ASCII, +96개 그래픽 문자 추가
- 유니코드: 32비트 문자 집합

- Java, C++ 와이드 문자에서 사용, ...
- 전 세계 대부분의 알파벳과 기호
- UTF-8, UTF-16: 가변 길이 인코딩

바이트/하프워드/단어 연산

- RISC-V 바이트/하프워드/워드 로드/저장
  - 바이트/하프워드/워드 로드: 부호가 64비트로 확장됩니다.
    - `LB RD,` 오프셋`(RS1)`
    - `LH RD,` 오프셋`(RS1)`
    - `LW RD,` 오프셋`(RS1)`
  - 부호 없는 바이트/하프워드/워드 로드: 0은 64비트로 확장됩니다.
    - `LBU RD,` 오프셋`(RS1)`
    - `LU RD,` 오프셋`(RS1)`
    - `LWU RD, OFFSET(RS1)`

- 바이트/하프워드/워드 저장: 가장 오른쪽 8/16/32비트 저장
  - `sb rs2,` 오프셋`(RS1)`
  - `sh rs2,` 오프셋`(RS1)`
  - `sw rs2,` 오프셋`(RS1)`

문자열 복사 예제

- C 코드:
  - 널로 끝나는 문자열

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

문자열 복사 예제

- RISC-V 코드:

```
strcpy:
     addi  sp,sp,-8          // 스택 조정          이중 단어 1개
     sd    x19,0(SP)         // push x19
     추가   x19,x0,x0         // i=0
L1:  추가   x5,x19,x10        // x5 = 주소          y[i]
     lbu   x6,0(x5)          // x6 = y[i]
     추가   x7,x19,x11        // x7 = 주소          x[i]
     sb    x6,0(x7)          // x[i] = y[i]
     beq   x6,x0,L2          // if y[i] == 0 를 클릭한 다음 종료
     addi  x19,x19, 1        // i = i + 1
     jal   x0,L1             // 루프의 다음 반복
L2:  ld    x19,0(SP)         // 저장된 X19 복원
     addi  sp,sp,8           // 스택에서 더블워드 1개 팝
     jalr  x0,0(x1)          // 를 클릭하고
```

32비트 상수

- 대부분의 상수는 작습니다.
  - 12비트 즉시로 충분
- 가끔 32비트 상수의 경우

  루이 `RD, 상수`
  - 20비트 상수를 rd의 비트 [31:12]로 복사합니다.
  - 비트 31을 비트 [63:32]로 확장합니다.

- rd의 비트 [11:0]을 0으로 지웁니다.

```
LUI X19, 976 // 0x003D0
```

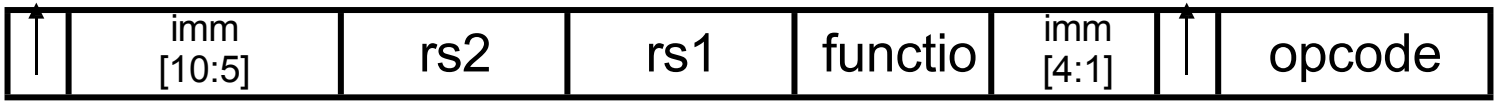
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

```
ADDI X19,X19,1280 // 0x500
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

지점 주소 지정

- 브랜치 지침은 다음을 지정합니다.
  - 옵코드, 레지스터 2개, 대상 주소
- 대부분의 지점 대상은 지점 근처에 있습니다.
  - 앞으로 또는 뒤로
- SB 형식:

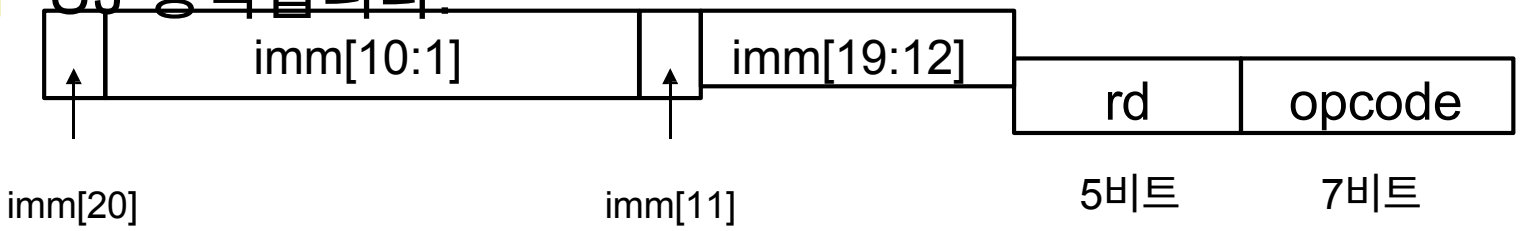


- PC 상대 주소 지정
  - 대상 주소 = PC + 즉시 × 2

(하프워드 경계: RISC-V는 2바이트 길이의 인스턴스 허용)

주소 지정 건너뛰기

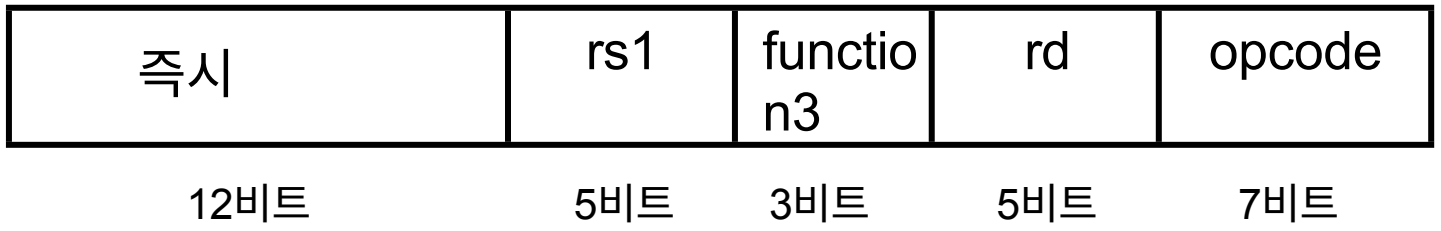
- 점프 및 링크(`jal`) 타겟은 더 넓은 범위를 위해 20비트 즉시를 사용합니다.
- UJ 형식입니다:



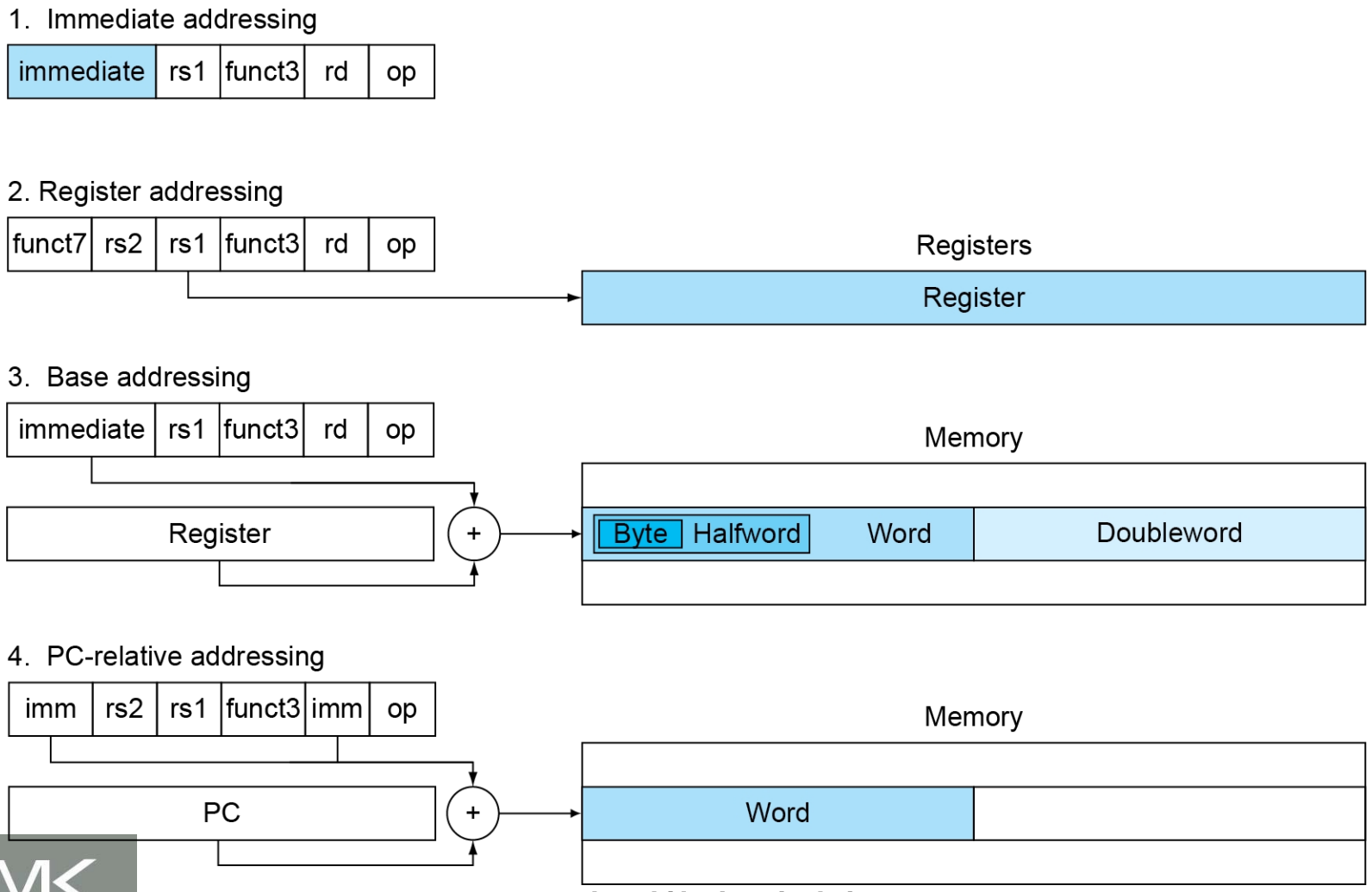
- 32비트 절대 주소로 긴 점프의 경우
  - 루이: 주소[31:12]를 임시 레지스터에 로드(상위 즉시 로

드)

- jalr: 주소[11:0]를 추가하고 대상으로 이동합니다(I 형식).



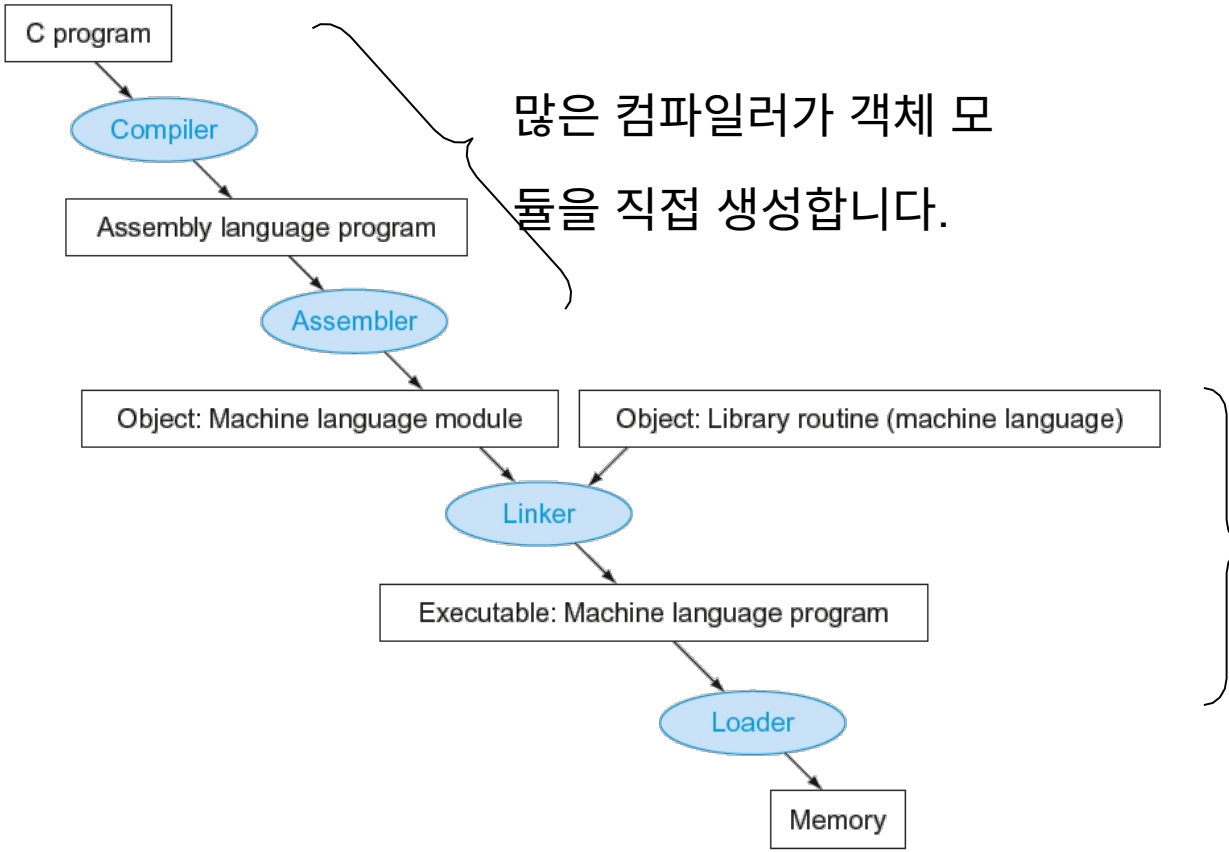
RISC-V 주소 지정 요약



RISC-V 인코딩 요약

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

번역 및 시작



많은 컴파일러가 객체 모듈을 직접 생성합니다.

정적 링크

객체 모듈 생성하기

- 어셈블러(또는 컴파일러)는 프로그램을 기계어 명령어로 변환합니다.
- 조각으로 완전한 프로그램을 구축하기 위한 정보를 제공합니다.
  - 헤더: 객체 모듈의 설명된 내용
  - 텍스트 세그먼트: 번역된 지침
  - 정적 데이터 세그먼트: 프로그램 수명 동안 할당된 데이

터

- 위치 정보: 로드된 프로그램의 절대 위치에 따라 달라지는 콘텐츠의 경우
- 심볼 테이블: 전역 정의 및 외부 참조
- 디버그 정보: 소스 코드와 연결하기 위한 정보

객체 모듈 연결

- 실행 가능한 이미지를 생성합니다.
  1. 세그먼트 병합
  2. 레이블 확인(주소 확인)
  3. 패치 위치 종속 및 외부 참조
- 재배치하는 로더가 수정할 위치 종속성을

남길 수 있습니다.

- 하지만 가상 메모리를 사용하면 이렇게 할 필요가 없습니다.
- 가상 메모리 공간의 절대 위치에 프로그램 로드 가능

프로그램 로드

- 디스크의 이미지 파일에서 메모리로 로드
  1. 헤더를 읽고 세그먼트 크기 결정
  2. 가상 주소 공간 만들기
  3. 텍스트 및 초기화된 데이터를 메모리에 복사
     - 또는 페이지 테이블 항목이 다음에서 결함이 발생하도록 설정합니다.

4. 스택에 인수 설정
5. 레지스터 초기화(SP, FP, GP 포함)
6. 시작 루틴으로 이동
   - 인수를 x10, ...에 복사하고 main을 호출합니다.
   - 메인이 반환되면 syscall을 종료합니다.

동적 연결

- 라이브러리 프로시저가 호출될 때만 링크/로드합니다.
  - 프로시저 코드가 재배치 가능해야 합니다.
  - 모든 (과도적으로) 참조된 라이브러리의 정적 링크로 인한 이미지 부풀림을 방지합니다.
  - 새 라이브러리 버전 자동으로 가져오기

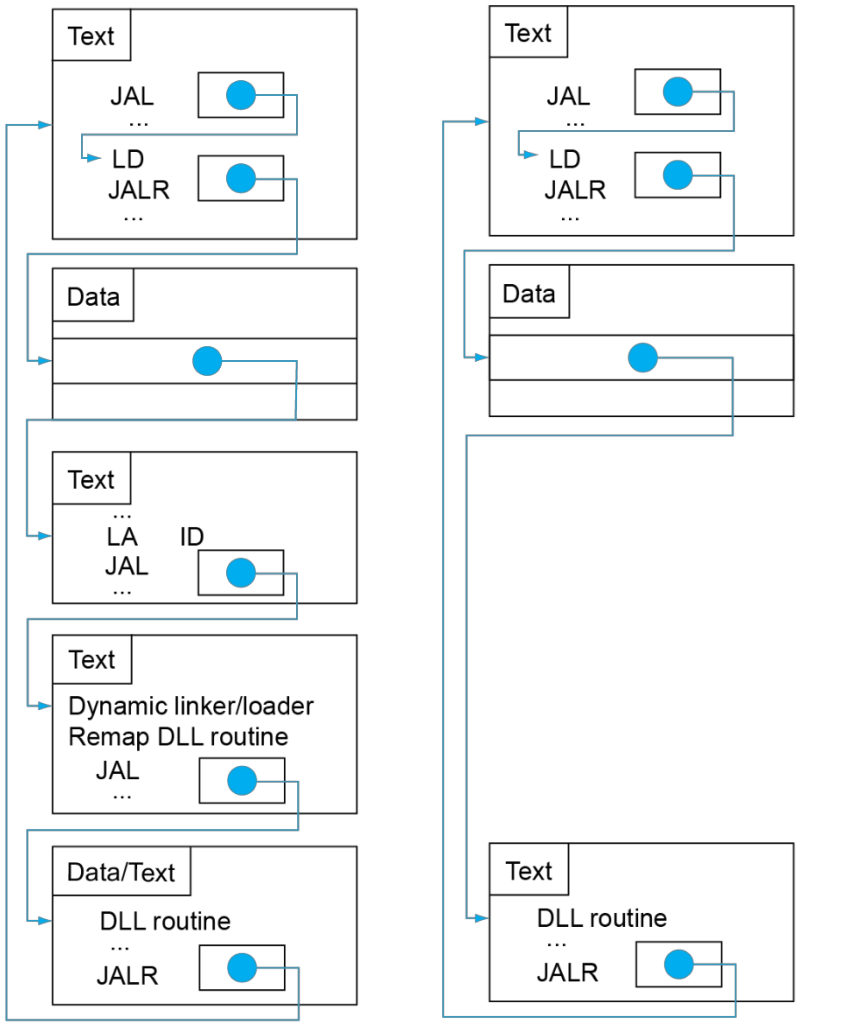
레이지 링크

방향 테이블

스텁: 루틴 ID를 로드합니다,

링커/로더로 이동하기 링커

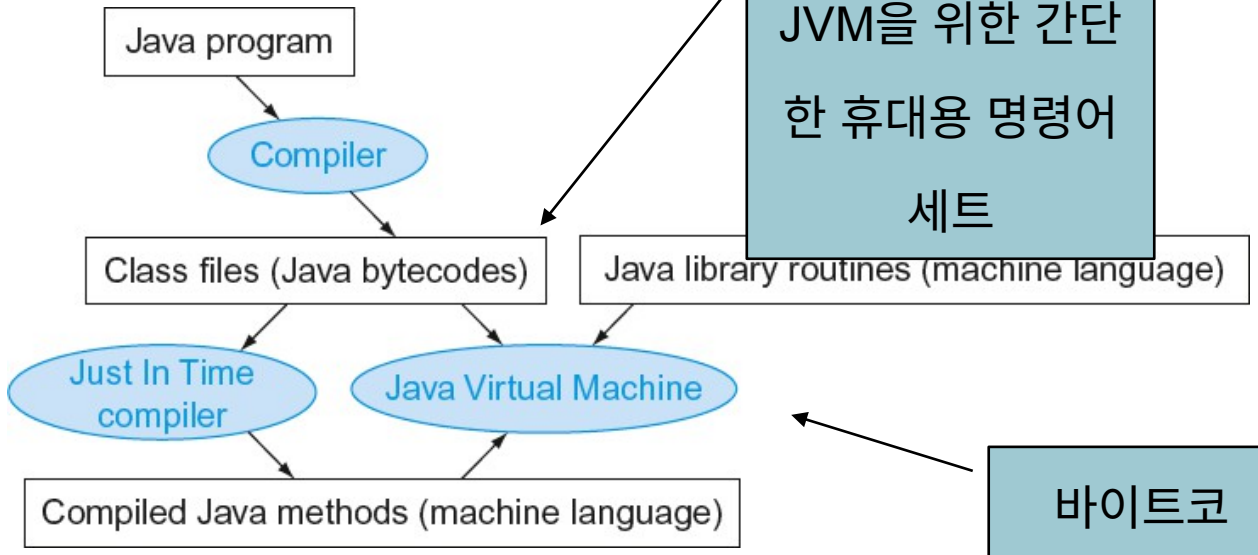
/로더 코드



(a) First call to DLL routine (b) Subsequent calls to DLL routine

동적으로 매핑된 코드

Java 애플리케이션 시작



JVM을 위한 간단한 휴대용 명령어 세트

"핫" 메서드의 바이트코드를 호스트 머신용 네이티브 코드로 컴파일합니다

바이트코드 해석

C 정렬 예제

- C 버블 정렬 함수에 대한 조립 지침 사용 예시
- 스왑 절차(리프)

```
void swap(long long int v[],
                  롱 롱 인트 K)
{
   롱 롱 인트 템프; 템프 =
   v[k];
```

```
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- V는 X10, K는 X11, TEMP는 X5

프로시저 스왑

스왑:

```
slli x6,x11,3     // reg x6 = k * 8
추가 x6,x10,x6    // reg x6 = v + (k * 8)
ld   x5,0(x6)     // reg x5 (임시) =  v[k]
ld   x7,8(x6)     // reg x7 = v[k + 1]
sd   x7,0(x6)     // v[k] = reg x7
sd   x5,8(x6)     // v[k+1] = REG X5 (TEMP)
jalr x0,0(x1)     // 반환    호출 루틴으로
```

C의 정렬 프로시저

- 비리프(통화 스왑)

```
void sort (long long int v[], size_t n)
{
  size_t i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

- X10의 V, X11의 N, X19의 I, X20의 J

외부 루프

- 외부 루프의 스켈레톤:
  - for (i = 0; i <n; i += 1) {

```
   li    x19,          0// i = 0
for1tst:
   BGE X19,X11,            EXIT1// X19 ≥ X11 (I≥N) IF EXIT1로 이
```

동합니다.

(외부 포루프의 본문)

```
   ADDI X19,X19,             1// I += 1
```

```
    j                       for1tst// 외부 루프 테스트 분기
exit1:
```

내부 루프

- 내부 루프의 골격:
  - for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j - = 1) {

```
      ADDI X20,X19,      -1// J = I -1
for2tst:
    blt  x20,x0,exit2   // X20 < 0(j < 0)이면 exit2로 이동합
                           니다.
    slli x5,x20,3       // REG X5 = J * 8
    추가 x5,x10,x5      // REG X5 = V + (J * 8)
    ld   x6,0(x5)       // reg x6 = v[j]
    ld   x7,8(x5)       // reg x7 = v[j + 1]
    ble  x6,x7,exit2    // 다음과 같은 경우 x6 ≤ x7
                           출구 2로 이동합니
                           다.
    mv   x21, x10       // 복사 매개변수 x10으로 x21
    mv   x22, x11       // 복사 매개변수 x11을 x22
```

```
mv   x10, x21        // 첫 번째 스왑 매개 변수는 v
mv   x11, x20        // 두 번째 스왑 매개변수는 J

   JAL X1,              SWAP// 통화 스왑
   addi x20,x20,        -1// j -= 1
   j                    for2tst// 내부 루프 출구 2를 테스트
```

하는 브랜치:

레지스터 보존

- 저장된 레지스터를 보존합니다:

```
 addi sp,sp,-40 // 스택에 5 레그를 위한 공간 확보
sd   x1,32(SP)  // 저장 스택에 x1
sd   X22,24(SP) // 저장 x22 on 스택
sd   X21,16(SP) // 저장 x21 on 스택
sd   x20,8(SP)  // 저장 x20 on 스택
sd   x19,0(SP)  // 저장 x19 on 스택
```

- 저장된 레지스터를 복원합니다:

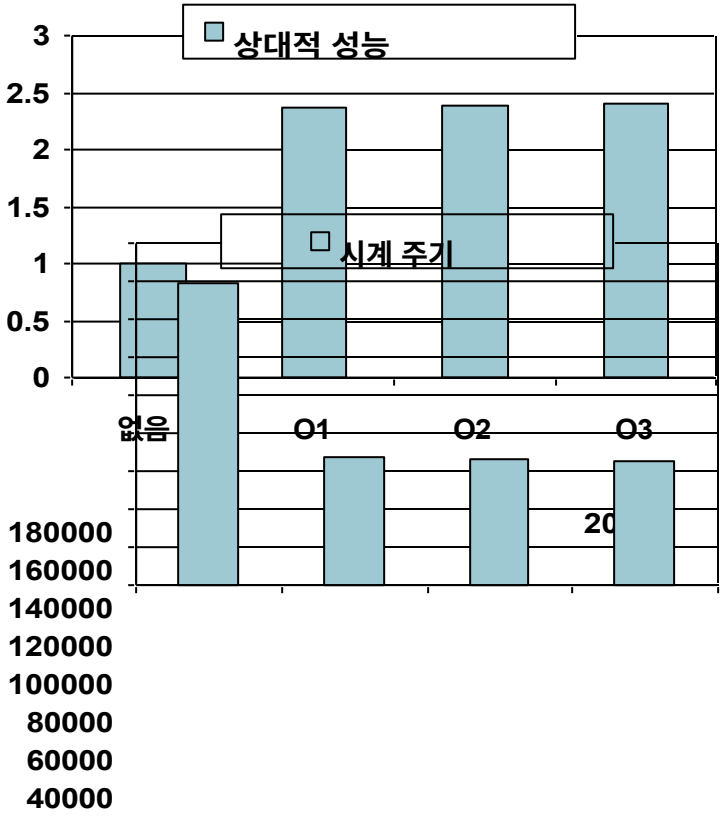
```
exit1:
  SDX19,0(SP) // 스택에서 X19 복원
   sd   x20,8(SP)  //   복원   x20 에서   스택
```

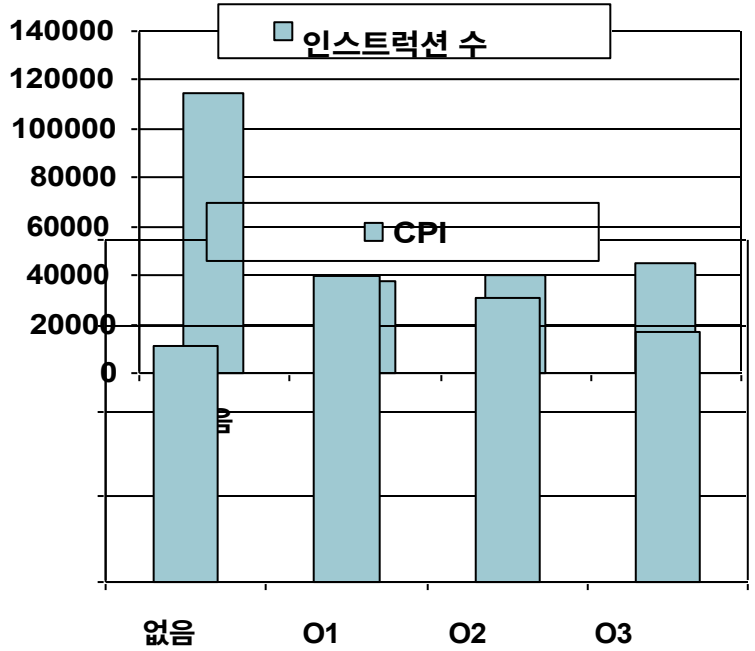
```
  sd    X21,16(SP) //
  sd    X22,24(SP) //
  sd    x1,32(SP)  //
  addi  sp, sp, 40 //
JALR X0,0(X1)
```

복원 x21 에서 스택
복원 x22 에서 스택
복원 스택에서 x1
복원 스택 포인터

컴파일러 최적화의 효과

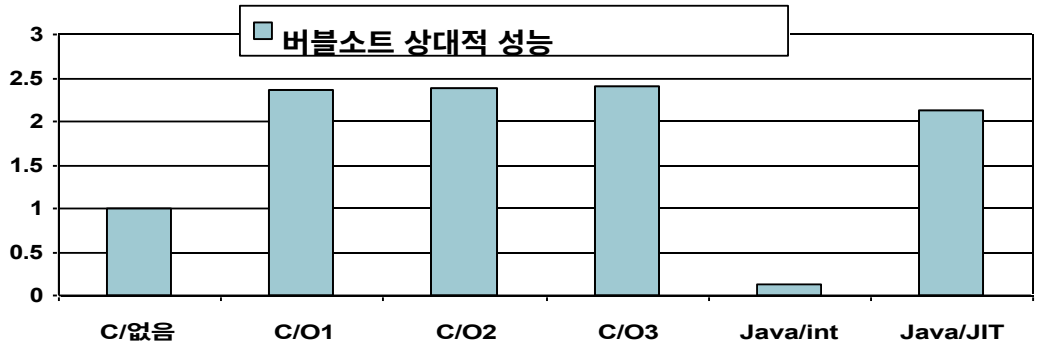
Linux에서 펜티엄 4용 gcc로 컴파일됨

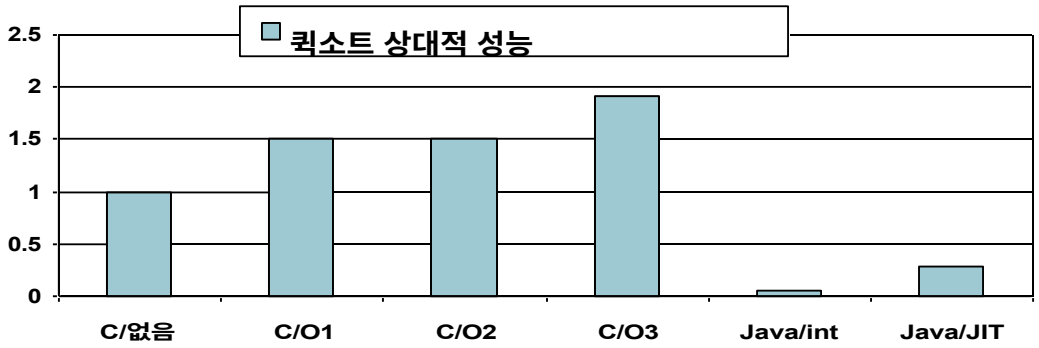


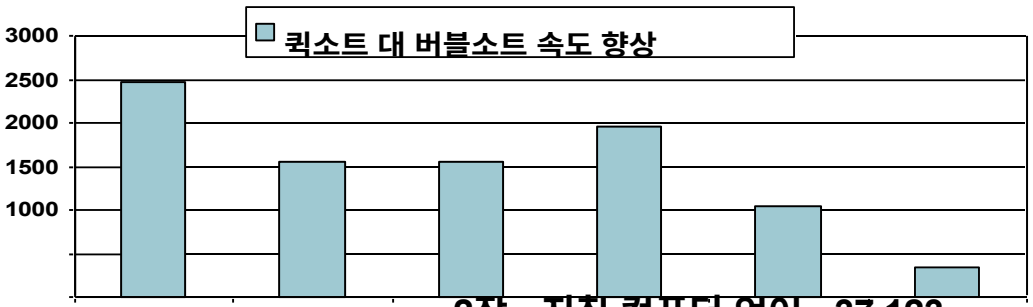



2
1.5
1
0.5
0
없음
O1
O2
O3

언어 및 알고리즘의 효과


버블소트 상대적 성능
3
2.5
2
1.5
1
0.5
0
C/없음
C/O1
C/O2
C/O3
Java/int
Java/JIT


퀵소트 상대적 성능
2.5
2
1.5
1
0.5
0
C/없음
C/O1
C/O2
C/O3
Java/int
Java/JIT


퀵소트 대 버블소트 속도 향상
3000
2500
2000
1500
1000


500
0
C/없음
C/O1
C/O2
C/O3
Java/int
Java/JIT

배운 교훈

- 명령어 수와 CPI는 따로 떼어놓고 보면 좋은 성능 지표가 아닙니다.
- 컴파일러 최적화는 알고리즘에 민감합니다.
- Java/JIT 컴파일된 코드는 JVM으로 해석된 코드보다 훨씬 빠릅니다.

- 경우에 따라 최적화된 C와 비슷합니다.

- 멍청한 알고리즘을 고칠 수 있는 것은 아무것도 없습니다!

배열 대 포인터

- 배열 인덱싱에는 다음이 포함됩니다.
  - 인덱스에 요소 크기 곱하기
  - 배열 기본 주소에 추가
- 포인터는 메모리 주소에 직접 대응합니다.
  - 인덱싱의 복잡성을 피할 수 있습니다.

예시: 배열 지우기

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
   li   x5,         0// i =
0 loop1:
   slli x6,x5,3     // x6 = i * 8
   addx7,x10,x6 // x7 = 주소
                    // of array[i]
   sd   x0,0(x7)    // array[i]
   =addi x5,x5,1    // i = i + 1
     blt x5,x11,loop1 // if (i<size)
                      // loop1로 이동
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
   MV X5,          X10// P = 주소
                   // of array[0]
   slli x6,x11,3 // x6 = size * 8
   add x7,x10,x6 // x7 = 주소
                   //의 배열[크기]
loop2:
   sd x0,0(x5)     //Memory[p]
   =addi x5,x5,8   // p = p + 8 bltu
   x5,x7,loop2
                   // if (p<&array[size])
```

```
// loop2로 이동
```

배열과 Ptr 비교

- "강도 감소"를 곱하여 교대
- 배열 버전은 시프트가 루프 내부에 있어야 합니다.
  - 증가된 i에 대한 인덱스 계산의 일부
  - 참조: 증분 포인터

- 컴파일러는 포인터를 수동으로 사용하는 것과 동일한 효과를 얻을 수 있습니다.
  - 유도 변수 제거
  - 프로그램을 더 명확하고 안전하게 만들기 위한 개선 사항

MIPS 지침

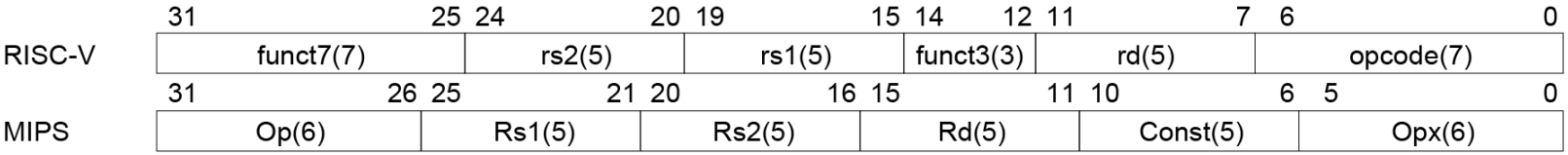
- MIPS: RISC-V의 상용 이전 버전
- 유사한 기본 지침 세트
  - 32비트 명령어
  - 32개의 범용 레지스터, 레지스터 0은 항상 0입니다.
  - 32개의 부동 소수점 레지스터
  - 로드/저장 명령에 의해서만 액세스되는 메모리
    - 모든 데이터 크기에 대해 일관된 주소 지정 모드 사용

다양한 조건부 분기

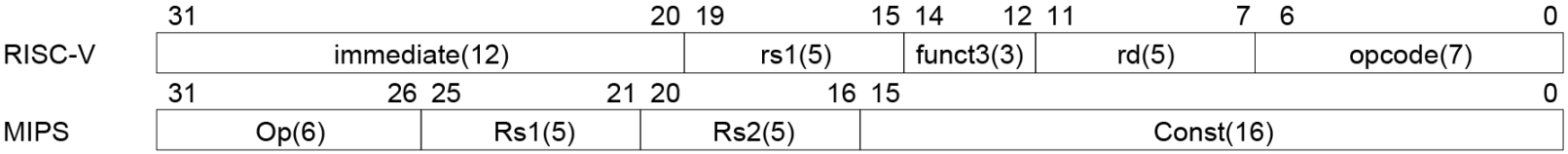
- <, <=, >, >=의 경우
- RISC-V: blt, bge, bltu, bgeu
- MIPS: 슬릿, 슬릿투(보다 작게 설정, 결과는 0 또는 1)
  - 그런 다음 beq, bne을 사용하여 브랜치를 완성합니다.

인스트럭션 인코딩

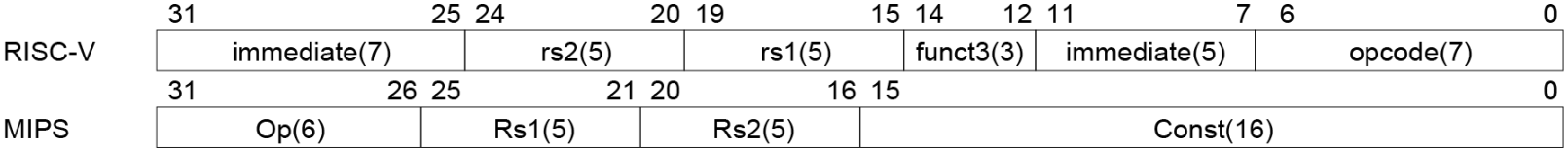
Register-register



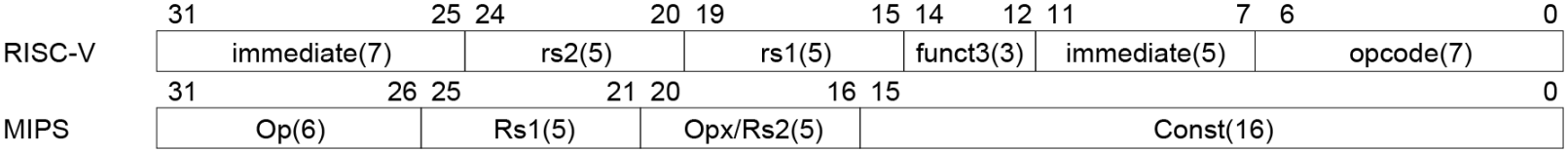
Load



Store



Branch



인텔 x86 ISA

- 이전 버전과의 호환성을 통한 진화
  - 8080 (1974): 8비트 마이크로프로세서
    - 어큐뮬레이터와 3개의 인덱스-레지스터 페어
  - 8086 (1978): 8080으로 16비트 확장
    - 복합 명령어 집합(CISC)
  - 8087(1980): 부동 소수점 코프로세서
    - FP 명령어 및 레지스터 스택 추가

- 80286 (1982): 24비트 주소, MMU
  - 세분화된 메모리 매핑 및 보호
- 80386 (1985): 32비트 확장(현재 IA-32)
  - 추가 주소 지정 모드 및 작업
  - 페이징 메모리 매핑 및 세그먼트

인텔 x86 ISA

- 더 진화...
  - i486(1989): 파이프라인, 온칩 캐시 및 FPU
    - 호환되는 경쟁사: AMD, Cyrix, ...
  - 펜티엄(1993): 슈퍼스칼라, 64비트 데이터 경로
    - 이후 버전에는 MMX(멀티미디어 확장) 지침이 추가되었습니다.
    - 악명 높은 FDIV 버그
  - 펜티엄 프로 (1995), 펜티엄 II (1997)

- 새로운 마이크로 아키텍처(콜웰, *펜티엄 연대기* 참조)

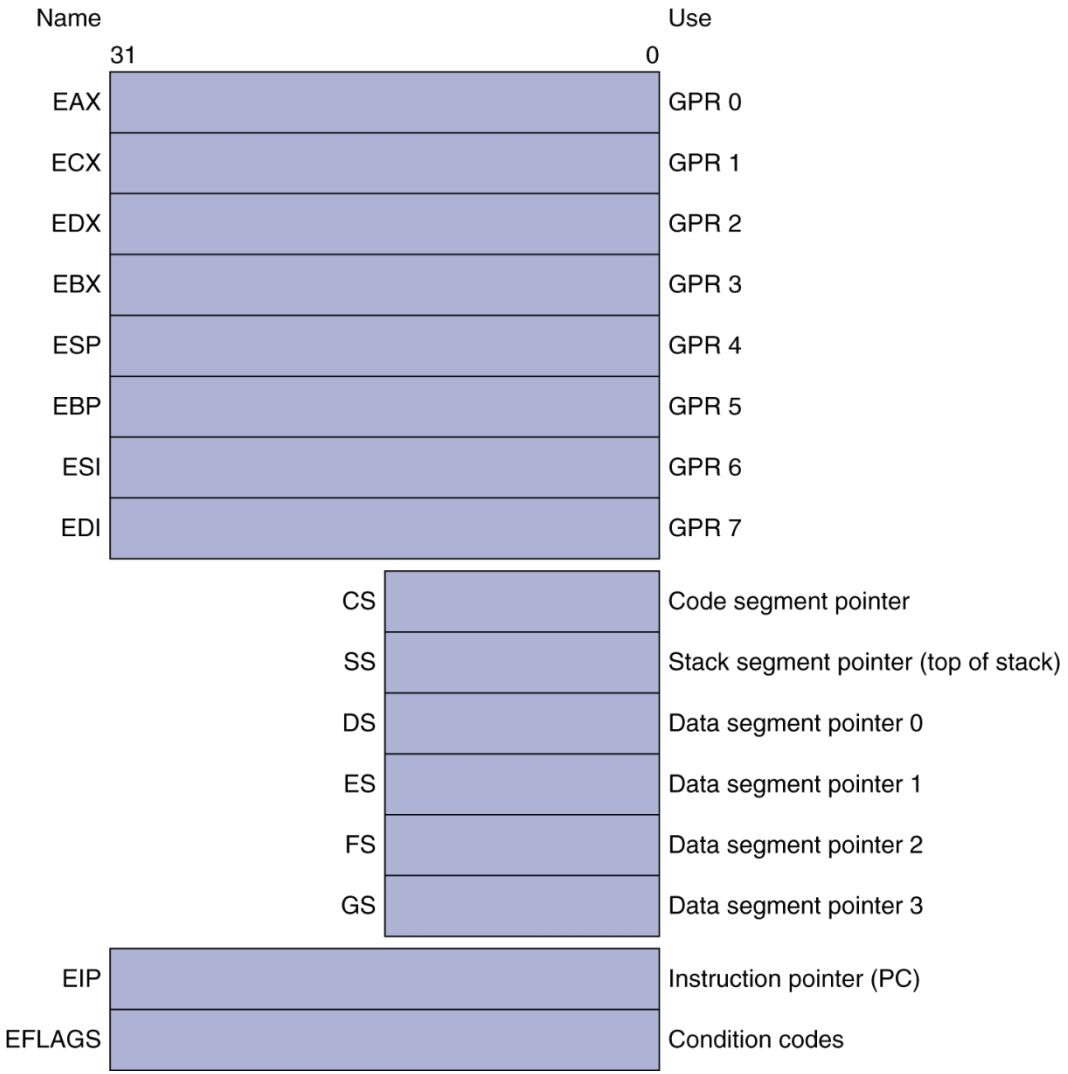
- 펜티엄 III (1999)
  - SSE(스트리밍 SIMD 확장) 추가 및 관련 레지스터
- 펜티엄 4 (2001)
  - 새로운 마이크로 아키텍처
  - SSE2 지침 추가

인텔 x86 ISA

- 그리고 더 나아가...
  - AMD64(2003): 64비트로 확장된 아키텍처
  - EM64T - 확장 메모리 64 기술(2004)
    - 인텔에서 채택한 AMD64(개선 사항 포함)
    - SSE3 지침 추가
  - 인텔 코어 (2006)
    - SSE4 지침, 가상 머신 지원 추가
  - AMD64(2007년 발표): SSE5 지침

- 인텔은 이를 따르지 않고 대신 ...
  - 고급 벡터 확장(2008년 발표)
    - 더 긴 SSE 레지스터, 더 많은 지침
- 인텔이 호환성을 확장하지 않으면 경쟁사가 확장할 것입니다!
  - 기술적 우아함 ≠ 시장 성공

기본 x86 레지스터


Name
Use
31
0
EAX
GPR 0
ECX
GPR 1
EDX
GPR 2
EBX
GPR 3
ESP
GPR 4
EBP
GPR 5
ESI
GPR 6
EDI
GPR 7
CS
Code segment pointer
SS
Stack segment pointer (top of stack)
DS
Data segment pointer 0
ES
Data segment pointer 1
FS
Data segment pointer 2
GS
Data segment pointer 3
EIP
Instruction pointer (PC)
EFLAGS
Condition codes

기본 x86 주소 지정 모드

- 명령어당 두 개의 피연산자

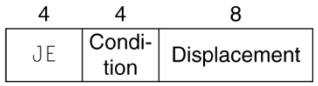
소스/대상 피연산자	두 번째 소스 피연산자
등록하기	등록하기
등록하기	즉시
등록하기	메모리
메모리	등록하기
메모리	즉시

- 메모리 주소 지정 모드

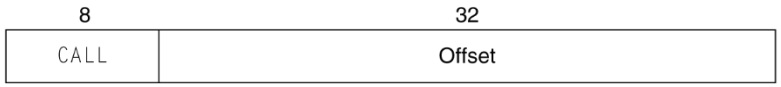
- 레지스터의 주소
- 주소 = R_{base} + 변위
- 주소 = R_{base} + 2scale × R_{index} (스케일 = 0, 1, 2, 3)
- 주소 = R_{base} + 2스케일 × R_{index} + 변위

x86 명령어 인코딩

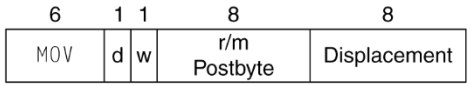
a. JE EIP + displacement



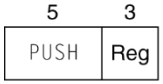
b. CALL



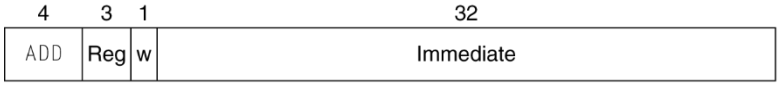
c. MOV EBX, [EDI + 45]



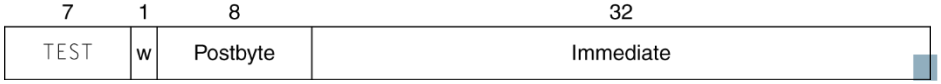
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- 가변 길이 인코딩
  - 후위 바이트는 주소 지정 모드를 지정합니다.
  - 접두사 바이트 수정

작업

- 피연산자 길이, 반복, 잠금, ...

IA-32 구현

- 복잡한 명령어 집합으로 구현이 어려운 경우
  - 하드웨어는 지침을 더 간단한 마이크로 작업으로 변환합니다.
    - 간단한 지침: 1-1
    - 복잡한 지침: 1-다수

- RISC와 유사한 마이크로엔진
- 시장 점유율을 통해 경제성 확보

- RISC와 비슷한 성능
  - 컴파일러는 복잡한 명령어를 피합니다.

기타 RISC-V 지침

- 기본 정수 명령어(RV64I)
  - 앞서 설명한 사항과 더불어
  - auipc rd, immed // rd = (imm<<12) + pc
    - 멀리뛰기 시 JALR(12비트 IMMED 추가)로 이어가기
  - SLT, SLTU, SLTI, SLTUI: 다음보다 작게 설정 (MIPS처럼)

- addw, subw, addiw: 32비트 add/sub
- SSLW, SRLW, SRLW, SSLIW, SRLIW, SRRAW: 32비트 시프트

- 32비트 변형: RV32I
  - 레지스터는 32비트 폭, 32비트 연산입니다.

명령어 세트 확장

- M: 정수 곱하기, 나누기, 나머지
- A: 원자 메모리 연산
- F: 단정밀도 부동 소수점
- D: 배정밀도 부동 소수점
- C: 압축된 지침

- 자주 사용하는 명령어를 위한 16비트 인코딩

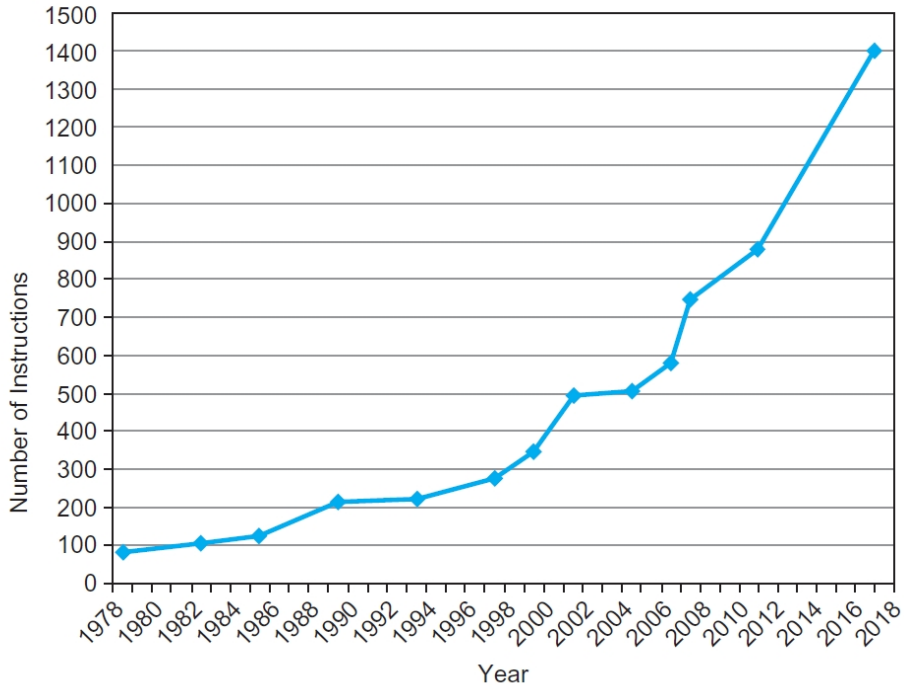
오류

- 강력한 명령어 ⇒ 더 높은 성능
  - 필요한 지침 수 감소
    - 하지만 복잡한 지침은 구현하기 어렵습니다.
- 간단한 명령어를 포함한 모든 명령어의 속도가 느려질 수 있습니다.
  - 컴파일러는 간단한 명령어에서 빠른 코드를 만드는 데 능숙합니다.
- 고성능을 위한 어셈블리 코드 사용

- 그러나 최신 컴파일러는 최신 프로세서를 더 잘 처리합니다.
- 코드 줄 수 증가 ⇒ 오류 증가 및 생산성 저하

오류

- 이전 버전과의 호환성 ⇒ 명령어 세트는 변경되지 않습니다.
  - 하지만 더 많은 지시를 내릴 수 있습니다.



x86 명령어 세트

함정

- 순차적 단어가 순차적 주소에 있지 않습니다.
  - 1이 아닌 4로 증가합니다!
- 프로시저가 반환된 후 자동 변수에 대한 포인터 유지

- 예를 들어, 인수를 통해 포인터를 다시 전달합니다.
- 스택이 터지면 포인터가 유효하지 않게 됩니다.

결론

- 디자인 원칙
  1. 단순함이 규칙성에 유리
  2. 작을수록 빠름
  3. 좋은 디자인에는 좋은 타협이 필요합니다
- 일반 케이스를 빠르게 만들기
- 소프트웨어/하드웨어 계층

- 컴파일러, 어셈블러, 하드웨어

- RISC-V: RISC ISA의 전형
  - C.F. x86