4장. 프로세서 구조

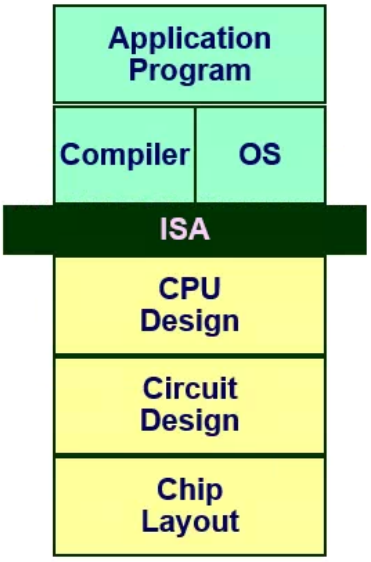
4.1 Y86-64 명령어 집합

4.2 논리 설계와 하드웨어 제어 언어 HCL
4.3 순차적 Y86-64 구현
4.4 파이프라이닝의 일반 원리
3.5 파이프라인형 Y86-64의 구현

4.1 Y86-64 명령어 집합

명령어 집합 구조 (Instruction Set Architecture, ISA)

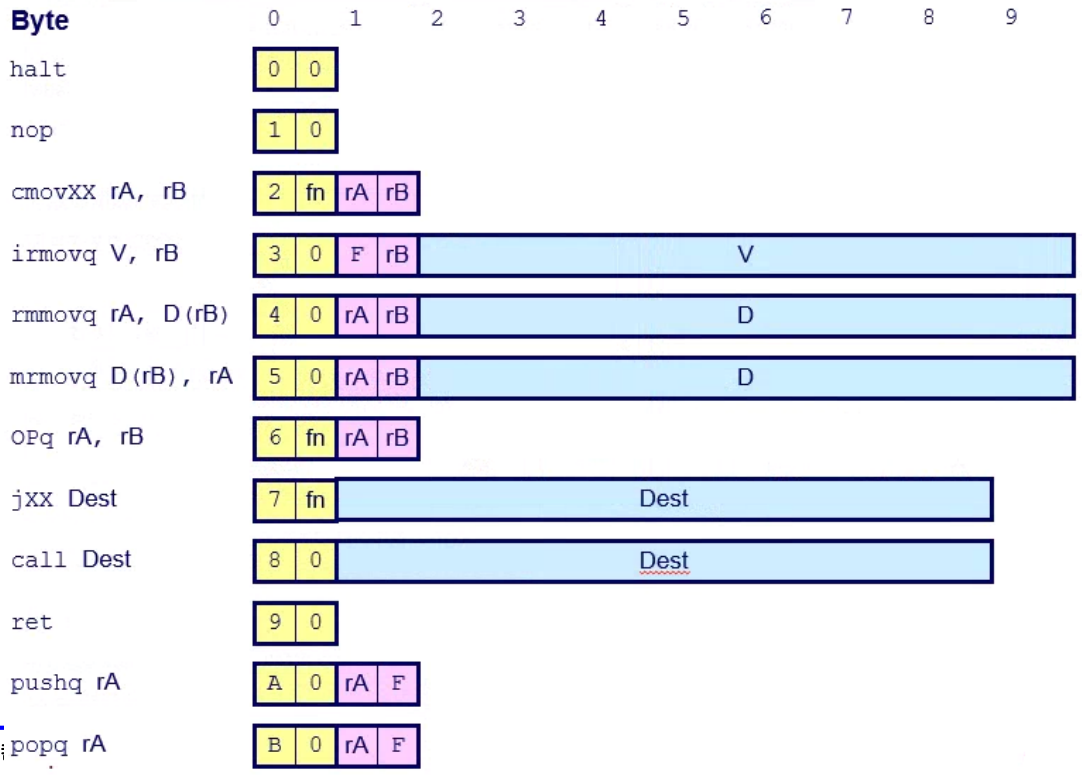
- 어셈블리 언어 관점
  - 프로그래머 관점
  - 프로세서 상태
    - 레지스터, 메모리, ...
  - 명령어
    - addq, pushq, ret, ...
    - 바이트로 인코드
- 추상화 계층
  - 상위 계층
    - 머신에서 프로그램하는 방법
    - 프로세서는 명령어들을 순차적으로 실행
  - 하위 계층
    - 머신의 구현
    - 성능 향상을 위해 다양한 기법들 적용
      - 예) 동시에 여러 명령어 실행



Y86-64 프로세서 상태

- Y86-64는 x86-64의 간략화 버전
- 프로그램 레지스터 (Program Registers)
  - 15개의 64비트 레지스터 (%r15 생략)
- 조건 코드 (Condition Codes)
  - 산술 또는 논리 연산에 의해 세트되는 단일 비트 플래그
    - ZF: Zero, SF: Sign, OF: Overflow
- 프로그램 카운터 (Program Counter)
  - 다음에 실행되는 명령어 주소를 가리킴
- 프로그램 상태 (Program Status)
  - 프로그램 실행의 정상 상태 또는 에러 상태 여부 표시
- 메모리 (Memory)
  - 바이트 단위로 주소 지정되고, 리틀 엔디안 방식으로 저장

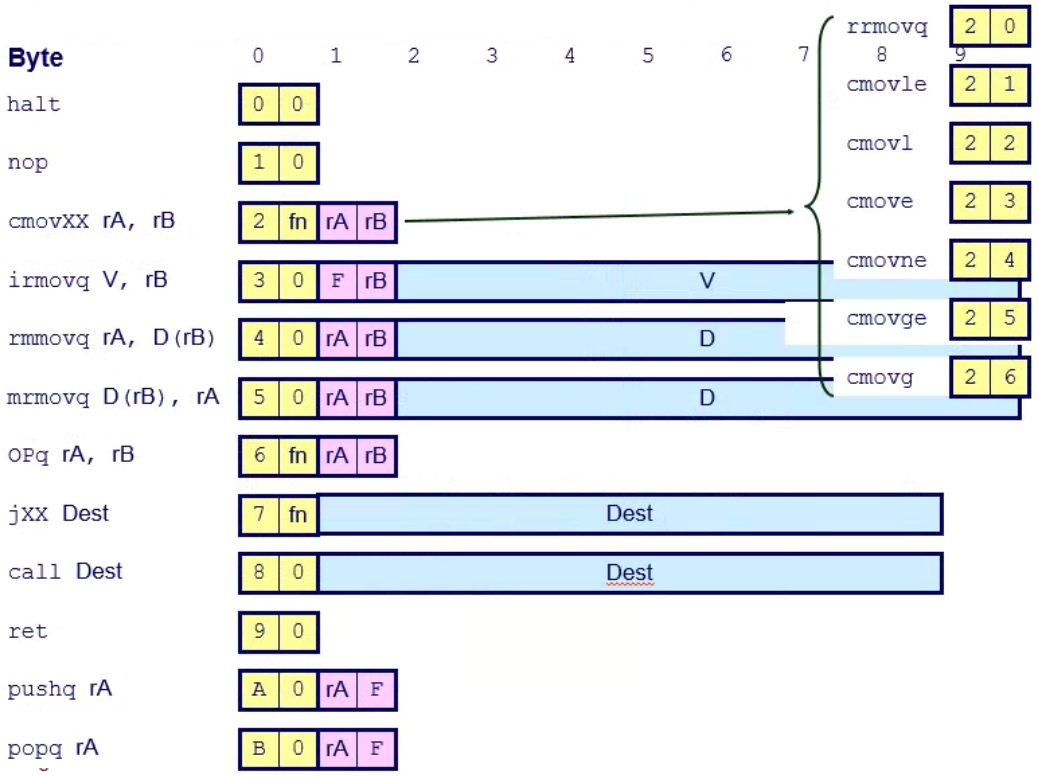
Y86-64 명령어 집합 (1)



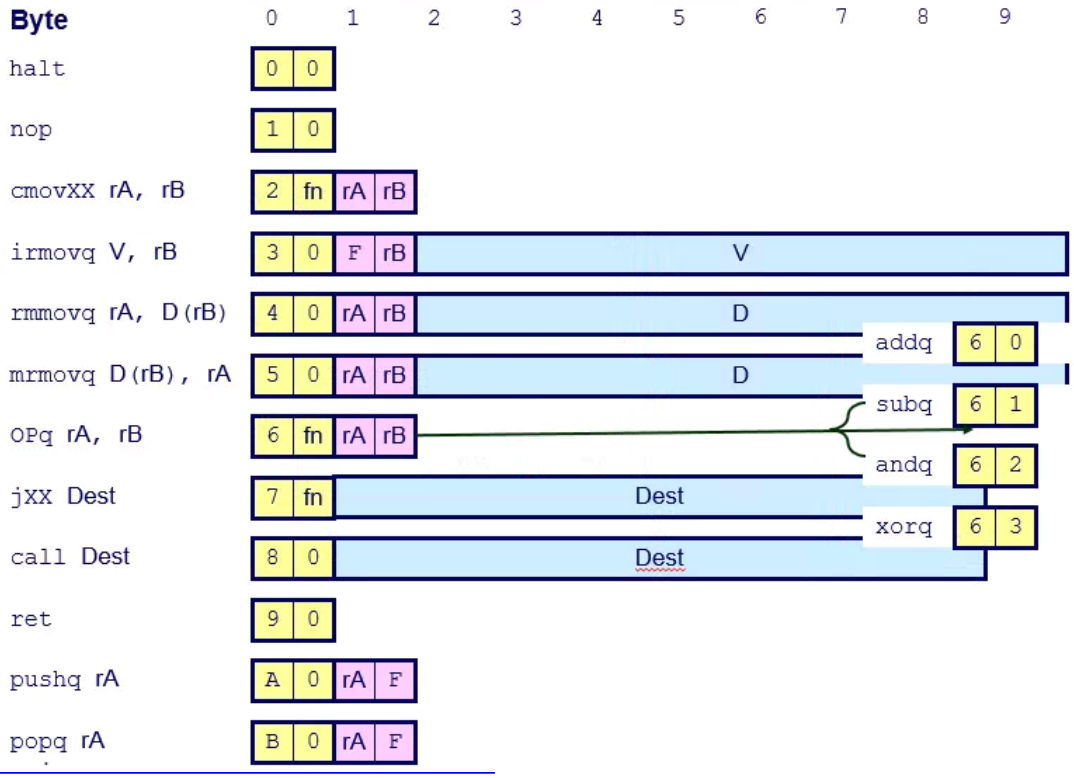
Y86-64 명령어 형식 (Instruction Format)

- 명령어 형식은 명령어 인코딩 방식을 표현
- 각 명령어는 필요한 필드에 따라 1에서 10바이트까지 사용
  - x86-64 보다도 적은 명령어 타입과 더 간단한 인코딩
- 첫 번째 바이트는 명령어 타입을 나타냄
  - 명령어 길이 결정
  - 2개의 4비트 필드

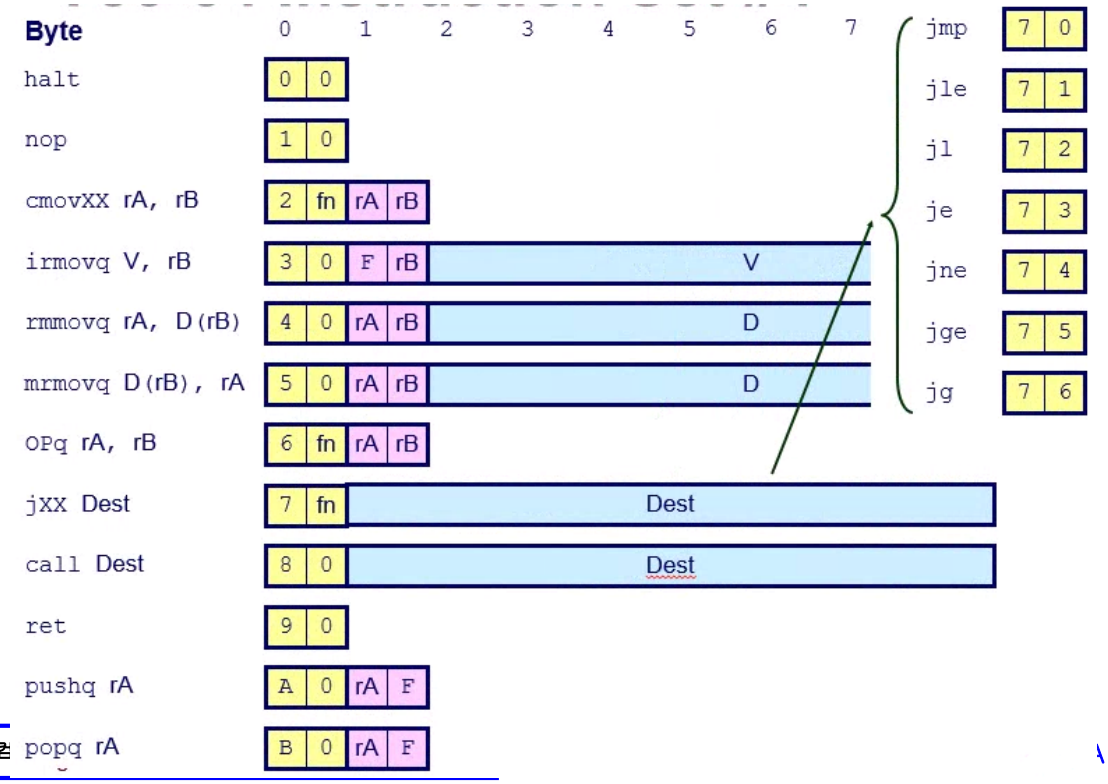
Y86-64 명령어 집합 (2)



Y86-64 명령어 집합 (3)



Y86-64 명령어 집합 (4)



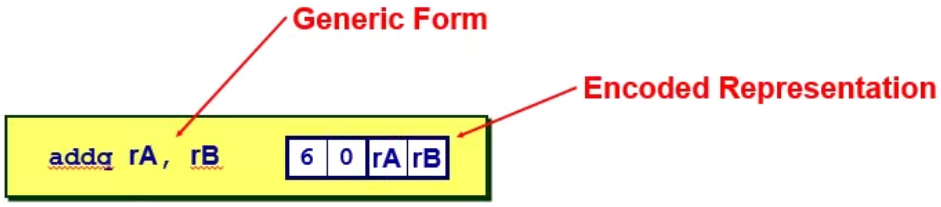
레지스터 인코딩

- 각 레지스터는 4 비트 ID로 표시
  - x86-64와 동일
  - 레지스터 ID 15 (0xF)는 “레지스터 없음”을 표시

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

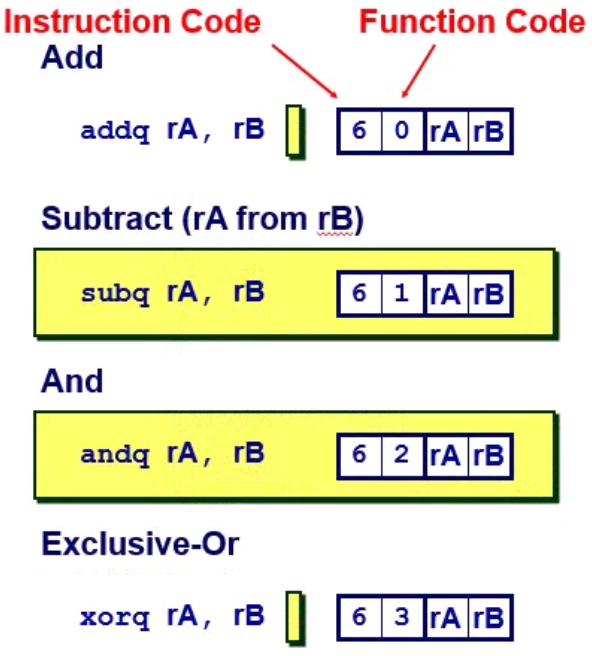
명령어 예

- 덧셈 명령어



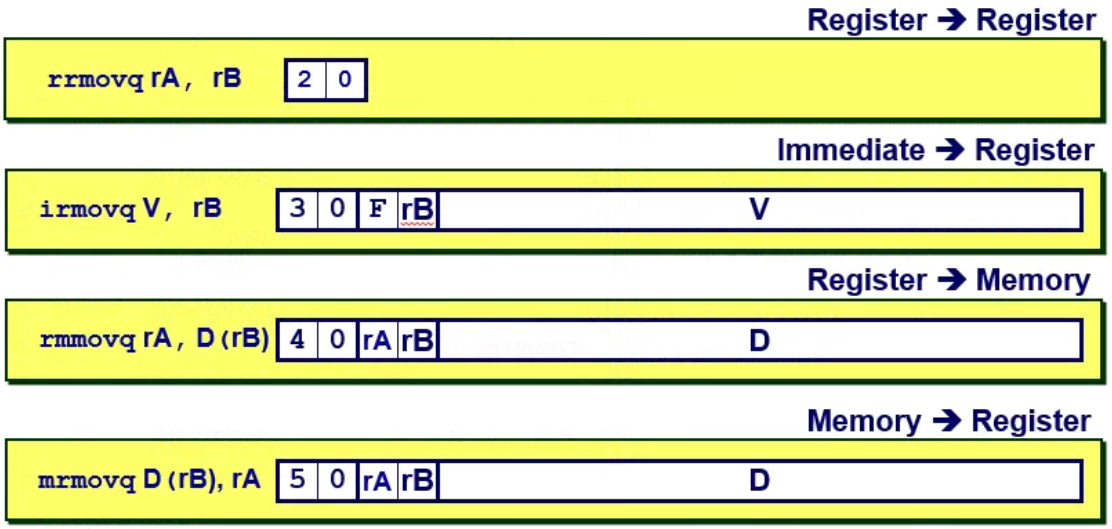
- rB <= rA + rB
- Y86-64는 레지스터 상의 데이터만 덧셈
- 덧셈 결과에 따라 조건 코드 세팅
- 예) addq %rax, %rsi 인코딩: 60 06
- 2 바이트 인코딩
  - 첫 번째 바이트 명령어 타입
  - 두 번깨 바이트는 소스와 목적지 레지스터

산술 및 논리 명령어



- Opq 로 표시
- 함수 코드 fn에 따라 인코딩이 달라짐
  - 첫 번째 바이트의 하위 4비트
- 연산 결과에 따라 조건 코딩 세팅

이동 명령어

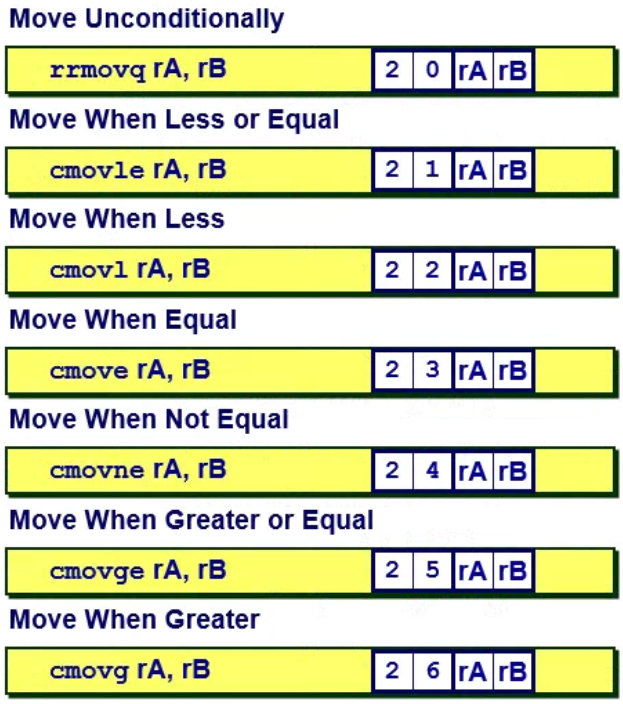


- x86-64의 movq 명령어와 유사
  - 더 간단한 메모리 주소 표현
  - 명령어 이름으로 오퍼랜드 표시

이동 명령어 예

X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
Encoding:	`30 82 cd ab 00 00 00 00 00 00`
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
Encoding:	`20 43`
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
Encoding:	`50 15 f4 ff ff ff ff ff ff ff`
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`
Encoding:	`40 64 1c 04 00 00 00 00 00 00`

조건 이동 명령어



- cmovXX 로 표시
- 함수 코드 fn에 따라 인코딩이 달라짐
- 조건 코드의 값에 기반하여 동작
- rrmovq 명령의 변형
  - 조건에 다라 소스에서 목적지 레지스터로 이동

점프 명령어 (1)

- jXX 로 표시
- 함수 코드 fn에 따라 인코딩이 달라짐
- x86-64의 해당 명령어와 유사함
- 완전한 목적지 주소를 인코딩
  - x86-64의 PC-상대 주소 지정방식과 다름

점프 명령어 (2)

Jump Unconditionally

jmp Dest	7	0	Dest

Jump When Less or Equal

jle Dest	7	1	Dest

Jump When Less

jl Dest	7	2	Dest

Jump When Equal

je Dest	7	3	Dest

Jump When Not Equal

jne Dest	7	4	Dest

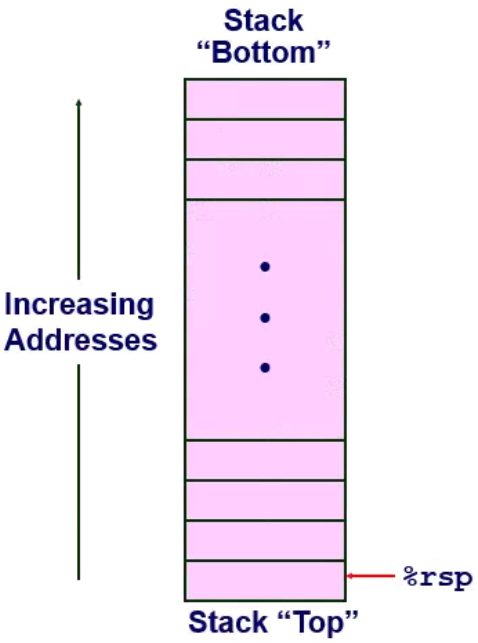
Jump When Greater or Equal

jge Dest	7	5	Dest

Jump When Greater

jg Dest	7	6	Dest

Y86-64 프로그램 스택



- 프로그램 데이터를 저장하는 메모리 영역
- 프로시저 호출 지원에 사용
- 스택의 탑은 %rsp가 가리킴
- 스택은 주소가 감소하는 방향으로 저장
  - push 시 먼저 스택 포인터가 감소
  - pop 후에 스택 포인터 증가

스택 명령어

- %rsp를 8만큼 감소
- rA를 %rsp가 가리키는 메모리에 저장
- x86-64와 유사

- rA를 %rsp가 가리키는 메모리로부터 읽음
- rA에 저장
- %rsp를 8만큼 증가
- x86-64와 유사

프로시저 호출 명령어

- 다음 명령의 주소를 스택에 push
- Dest가 가리키는 명령어의 실행을 시작
- x86-64와 유사

- 스택에서 리턴 주소를 pop
- 다음에 실행될 명령으로 리턴 주소를 사용
- x86-64와 유사

기타 명령어

- 어떤 동작도 수행하지 않음 (no operation)

- 명령어의 실행을 중지
- 86-64도 비슷한 명령어가 있지만, 사용자 모드에서는 실행할 수 없음
- 시뮬레이터 중지 시 사용
- 인코딩이 0이어서 0으로 초기화된 메모리가 실행 시에는 중지(halt) 명령어 실행

상태 코드 (Status Code)

- 프로그램의 전체적인 상태를 표시

Mnemonic	Code
AOK	1

■ **Normal operation**

Mnemonic	Code
HLT	2

■ **Halt instruction encountered**

Mnemonic	Code
ADR	3

■ **Bad address (either instruction or data) encountered**

Mnemonic	Code
INS	4

■ **Invalid instruction encountered**

- 바람직한 동작
  - AOK이면 계속 실행
  - 아니면 프로그램 실행 중지

sum() 함수 예 – C 프로그램

- start가 가리키는 메모리 주소에서부터 count 만큼의 8 바이트 값(long int)을 누적 더하는 프로그램

```
long sum(long *start, long count)
{
    long sum = 0;

    while (count) {
        sum += *start;
        start++;
        count--;
    }
    return sum;
}
```


sum() 함수 예 – x86-64 어셈블리 프로그램

- x86-64 머신의 GCC 컴파일러의 -Og 옵션으로 생성된 어셈블리 코드

```
# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:
    movl $0, %eax           # sum = 0
    jmp  .L2                # Goto test
.L3:                        # loop:
    addq (%rdi), %rax       # sum += *start;
    addq $8, %rdi           # start++
    subq $1, %rsi           # count--
.L2:
    testq %rsi, %rsi        # if (count != 0)
    jne   .L3               #  Goto loop
    rep ret                 # Return
```

sum() 함수 예 – Y86-64 어셈블리 프로그램

- Y86-64 머신의 어셈블리로 작성된 프로그램

```
# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:
        irmovq $8,%r8          # Constant 8
        irmovq $1,%r9          # Constant 1
        xorq %rax,%rax         # sum = 0
        andq %rsi,%rsi         # Set CC
        jmp    test            # Goto test
loop:
        mrmovq (%rdi),%r10     # sum += *start;
        addq %r10,%rax
        addq %r8,%rdi          # start++
        subq %r9,%rsi          # count--,  Set CC
test:
        jne    loop            # Stop when 0
        ret                    # Return
```


sum() 함수 예 – 어셈블리 코드 비교

- x86-64 코드와 Y86-64 코드 비교
  - Y86-64 코드의 산술 연산에서 즉시값(상수)를 사용할 수 없기 때문에 상수들을 레지스터에 적재하는 코드 추가
    - Y86-64 코드에서는 산술 및 논리 연산에서 오퍼랜드는 레지스터 값
  - Y86-64 코드의 산술 연산의 오퍼랜드는 레지스터이어야 하기때문에 메모리에서 오퍼랜드의 값을 읽어들이는 코드 추가
  - 손으로 작성한 Y86-64 코드에서 루프 분기 시에 조건 코드를 사용하여 testq 명령을 제거

Y86-64 어셈블리 프로그램 구조 (1)

```
init:                   # Initialization
   . . .
   call Main
   halt

   .align 8             # Program data
array:
   . . .

Main:                   # Main function
   . . .
   call sum    . . .

sum:                    # sum function
   . . .

   .pos 0x200           # Placement of stack
Stack:
```

- 프로그램은 주소 0부터 시작
- 스택을 설정해야 함
  - 스택의 위치 (주소)
  - 스택 포인터 (%rsp) 값 초기화
- 데이터를 초기화해야 함

Y86-64 어셈블리 프로그램 구조 (2)

```
init:
      # Set up stack pointer
      irmovq Stack, %rsp
      # Execute main program
      call Main
      # Terminate
      halt

# Array of 5 elements
      .align 8
Array:
      .quad 0x000d000d000d000d
      .quad 0x00c000c000c000c0
      .quad 0x0b000b000b000b00
      .quad 0xa000a000a000a000
      .quad 0x700080009000a000
```

- 프로그램은 주소 0부터 시작
- 스택을 설정해야 함
- 데이터를 초기화해야 함
- 변수의 주소로 심볼 이름 사용

Y86-64 어셈블리 프로그램 구조 (3)

```
Main:
        irmovq array,%rdi
        irmovq $5,%rsi
        # call sum(array,5)
        call sum
        ret
```

- sum 호출 설정
  - x86-64 프로시저 호출 규칙을 따름
  - array 주소와 count 값을 인수로 전달

과제 4-1: sum() 함수의 실행 및 분석 (실습과제)

- 가상머신에서 앞의 sum() 함수와 이를 호출하는 메인 함수를 작성하여 실행하고 분석
  - x86-64 머신 상에서 C 프로그램을 작성하고 컴파일 한 후 gdbgui로 실행
    - 어셈블리 소스 분석 및 실행 상태 추적
  - Y86-64 머신 상에서 어셈블리 프로그램을 작성
    - YAS 어셈블로 어셈블하여 생성된 목적 코드 분석
    - YIS 시뮬레이터로 실행하여 결과 분석
    - 실습 강의노트 3 참조
  - 위 두 어셈블리 코드의 차이점 분석

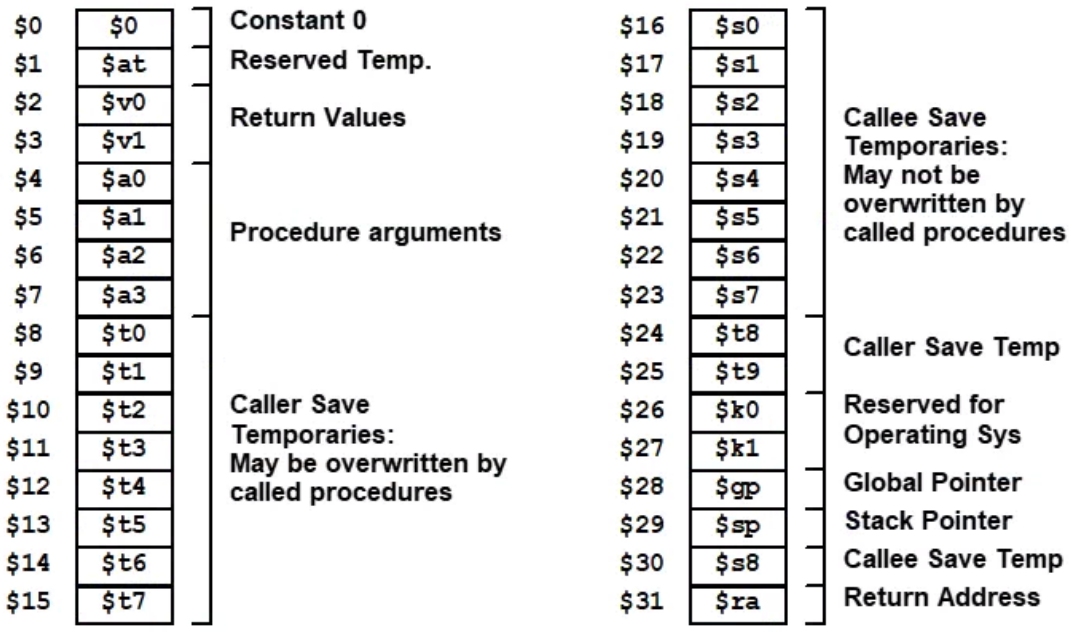
CISC 명령어 집합

- CISC (Complex Instruction Set Computer)
  - IA32가 대표적인 CISC
- 스택-중심의 프로시저 호출
  - 인수 전달 및 리턴 주소 저장에 스택을 사용
- 산술 및 논리 연산에 메모리 접근 가능
  - addq %rax, 12(%rbx, %rcx,8)
    - 메모리 읽기와 쓰기가 필요
    - 복잡한 주소 계산
- 조건 코드
  - 산술 및 논리 연산의 결과로 조건 코드 세트

RISC 명령어 집합

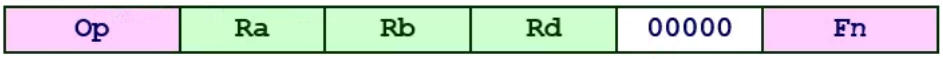
- RISC (Complex Instruction Set Computer)
  - 초창기 IBM에서 시작
  - 스탠포드 대학의 Hennesy와 버클리 대학의 Patterson이 완성
- 적은 수의 간단한 명령어
  - 작고 빠른 하드웨어로 명령어 실행
- 레지스터-기반 명령어 집합
  - 많은 레지스터 (보통 32개)
  - 레지스터에 인수, 리턴 주소, 임시값 등을 저장
- load와 store 명령만이 메모리 접근
  - Y86-64의 mrmovq와 rmmovq외 유사
- 조건 코드가 없음
  - test 명령의 결과가 0/1로 레지스터에 저장

RISC 예 – MIPS 레지스터



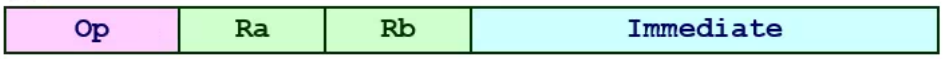
RISC 예 – MIPS 명령어

R-R



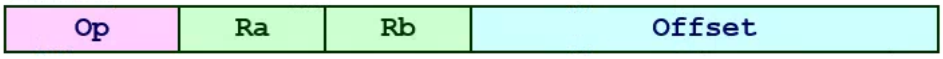

```
addu $3,$2,$1        # Register add: $3 = $2+$1
```

R-I



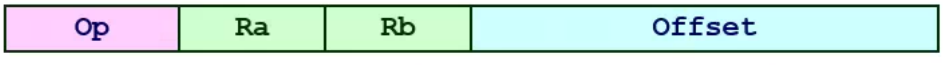

```
addu $3,$2, 3145     # Immediate add: $3 = $2+3145
sll $3,$2,2          # Shift left: $3 = $2 << 2
```

Branch




```
beq $3,$2,dest       # Branch when $3 = $2
```

Load/Store




```
lw $3,16($2)         # Load Word: $3 = M[$2+16]
sw $3,16($2)         # Store Word: M[$2+16] = $3
```

CISC와 RISC

- 초기 논쟁
  - CISC 측
    - 컴파일러 개발이 용이하고, 코드의 길이가 작음
  - RISC 측
    - 컴파일러 최적화가 쉽고, 간단한 칩 설계로 빠르게 수행
- 현재 상태
  - 데스크탑 프로세서에서는 ISA의 선택이 중요하지 않음
    - 하드웨어가 충분해서 양쪽 다 빠르게 수행
    - 코드 호환성이 더 중요
  - x86-64가 RSIC의 많은 장점을 수용
    - 레지스터 수를 늘리고, 인수 전달에 사용
  - 임베디드 프로세서인 경우 RISC가 적합
    - 작고, 값싸고, 전력 소모가 적음
    - 대부분의 스마트폰이 ARM 프로세서를 사용

요 약

요 약

- Y86-64 명령어 집합 구조
  - x86-64와상태와 명령어가 유사
  - 더 간단한 명령어 인코딩
  - CISC와 RISC 중간
- ISA 설계의 중요성
  - 현재는 하드웨어 용량이 충분하여 대부분 빠르게 수행하므로 이전보다 중요성이 덜해짐

과 제

과제 4-1: sum() 함수의 실행 및 분석 (실습과제)

- 가상머신에서 앞의 sum() 함수와 이를 호출하는 메인 함수를 작성하여 실행하고 분석
  - x86-64 머신 상에서 C 프로그램을 작성하고 컴파일 한 후 gdbgui로 실행
    - 어셈블리 소스 분석 및 실행 상태 추적
  - Y86-64 머신 상에서 어셈블리 프로그램을 작성
    - YAS 어셈블로 어셈블하여 생성된 목적 코드 분석
    - YIS 시뮬레이터로 실행하여 결과 분석
    - 실습 강의노트 3 참조
  - 위 두 어셈블리 코드의 차이점 분석